WESTERN DIGITAL
C O R P O R A T I O N

Western Digital UCSD Pascal™ III.0 Operating System Reference Manual

Release A0
July 1982

Document Internal Number: 80-013016-00A0
Manual Product Number
(Use for Ordering): WD9893

The MicroEngine Company

a wholly owned subsidiary of: WESTERN DIGITAL CORPORATION

2445 McCABE WAY, IRVINE, CALIFORNIA 92714 • (714) 557-3550 • TWX 910-595-1139

WARNING

OPERATION OF THIS DEVICE IN A RESIDENTIAL AREA MAY CAUSE HARMFUL INTERFERENCE REQUIRING THE USER TO TAKE WHATEVER STEPS MAY BE NECESSARY TO CORRECT THE INTERFERENCE.

60-013041-10

First Printed: July 1982

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

This manual is one in a series of publications that support the Western Digital Corporation UCSD Pascal(TM) III.Ø Operating System. This operating system, UCSD Pascal(TM), was developed at the University of California, San Diego, but has been enhanced and refined by The MicroEngine Company (a subsidiary of Western Digital Corporation).

This manual is not tutorial in nature and does not describe the operational aspects of the software. However, the first four chapters are written to provide new users with a quick grasp of the system command structure, the Editors, and the Filer. This book is intended to be a reference manual for users of the III.Ø Operating System.

ORGANIZATION OF THIS MANUAL

This reference manual is divided into seven main chapters and ten appendices.

- Chapter 1 provides an overview of the III.Ø Operating System including a general explanation of the outer and inner command levels.
- Chapter 2 contains discussions of some system fundamental concepts - namely, files and volumes.
- Chapter 3 describes four system editors: The Screen-Oriented Editor, the Advanced Editor, the L2 Editor, and YALOE (Yet Another Line-Oriented Editor).
- Chapter 4 discusses the system File Handler (Filer).
- Chapter 5 describes the Pascal Compiler.
- Chapter 6 provides information on numerous system utility programs that are part of the III.Ø Operating System.
- Chapter 7 explains some Pascal programming considerations for using the III.Ø Operating System on ME16ØØ and SB16ØØ computer systems.
- Appendix A lists command summaries for the outer level of commands, the Screen-Oriented Editor, the Line-Oriented Editor (YALOE), the Filer, and the Pascal Compiler.
- Appendix B contains several tables of information:

  Run-Time Errors
  I/O Results
  Syntax Errors
  Unit Numbers

- Appendix C contains tables of the P-machine opcodes, operator execution times, and the opcodes in a Pascal-like Metalanguage.
- Appendix D is an ASCII code chart.
- Appendix E lists the UCSD Pascal(TM) reserved words.
- Appendix F shows UCSD syntax diagrams.
- Appendix G contains tables of ME16ØØ and SB16ØØ I/O addresses.
- Appendix H lists the ME16ØØ and SB16ØØ boot and initialization diagnostic messages.
- Appendix I gives the code for the system globals of the H3 release of the III.Ø Operating System.
- Appendix J describes the hardware and software changes for the operating system from versions GØ to H3.
- Appendix K is a glossary of terms.

Please submit the Publication Comment Form (located at the back of this manual) with any comments about this document to:

THE MICROENGINE COMPANY
Subsidiary of Western Digital Corp.
2445 McCabe Way
Irvine, California 92714
ATTN: Product Documentation

*****

This guide was prepared and edited using the
Western Digital Pascal(TM) Screen-Oriented Editor.

RELATED DOCUMENTS

The following publications provide additional information on 16ØØ Series SuperMicro Computer Systems.

- ME16ØØ Modular Series SuperMicro Computer System Installation/Operation Guide (order number ME169Ø).

  This book provides basic information for setting up a new ME16ØØ computer system.

- MICROENGINE(TM) Computer Systems Peripheral Device Configuration Guide (order number ME1692).

  This guide describes procedures for configuring nonstandard peripheral devices to be installed with 16ØØ series systems.

- Getting Started with the Western Digital 16ØØ Series SuperMicro Computer System (order number ME1694).

  This document briefly describes the major components of the III.Ø Operating System and steps the user through one simple session using the system. It is included as part of the set of documents to accompany the 16ØØ product line.

- SB16ØØ Series SuperMicro Computer System Installation/Operation Guide (order number SB169Ø).

  This book provides basic information for setting up a new SB16ØØ computer system.

- Introduction to Pascal Including UCSD Pascal by Rodney Zaks (order number WD9891).

  This book introduces the reader to the Pascal programming language.

- Beginner's Guide for the UCSD Pascal System by K. Bowles (order number WD9892).

  This tutorial book aids in understanding and gaining familiarity with the UCSD Pascal(TM) operating system.

For copies of these documents, see your Western Digital sales representative.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

1. OVERVIEW OF THE III.Ø OPERATING SYSTEM

The Western Digital UCSD Pascal(TM) III.Ø Operating system is designed to run on the ME16ØØ and SB16ØØ SuperMicro Series Computer System produced by The MicroEngine Company, subsidiary of Western Digital Corporation. This operating system is an enhanced version of the UCSD Pascal(TM) III.Ø Operating System.

Basically a single-user program development system, the III.Ø Operating System provides a complete environment for both program development and execution and text processing. Comprised of several modules plus numerous utility programs, the III.Ø Operating System is a multitasking operating system.

This operating system allows multiple tasks to run concurrently based on priority.

The major components of the III.Ø Operating System are listed below and are discussed in this manual; also, a chapter pertaining to Pascal programming is included.

- System Filer.
- System Editors (Screen-Oriented, L2, Advanced, and YALOE).
- System Compiler (UCSD Pascal).
- System Utilities.

The overall structure of the III.Ø Operating System in regard to commands is discussed in this chapter.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

1.1 III.Ø OPERATING SYSTEM COMMAND STRUCTURE

The III.Ø Operating System command structure is composed of an outer level of commands and an inner level of commands. The outer level of commands allows access to the inner level of commands or enables performance of specific functions. For example, typing an E from the outer level command line accesses the Screen-Oriented Editor, which, in turn, displays a command prompt line (inner level of commands); or typing an X from the outer level command line executes a code file (performance of a specific function).

The relationship of the outer level of commands and the inner levels of commands is shown in Figure 1-1.

The following two subsections discuss the outer level of commands and the inner level of commands, respectively.

OUTER LEVEL COMMANDS

Command: E(dit, R(un, F(ile, C(omp, L(ink, X(ecute, D(ebug ?

? = Command: A(da, U(ser restart, I(nitialize, H(alt

E

F

INNER LEVEL COMMANDS

> Edit: A(djt C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap ?

? = >move[<arrow>,<sp>,<ret>,=, P(age], direction[<,>], M(rgn, S(et, V(rfy

INNER LEVEL COMMANDS

Filer: G(et, S(ave, W(hat, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

? = Filer: B(ad-blks, E(xt-dir, K(rnch, M(ake, P(refix, V(als, X(amine, Z(ero

NOTE

Typing a "?" causes the second (unseen) command prompt line to be displayed.

Figure 1-1. III.Ø Operating System Command Structure.

1.2 OUTER LEVEL OF COMMANDS

The outer level of commands is automatically displayed across the top of the screen in the following three cases: (1) after booting or automatic execution of the operating system; (2) after any of the lower levels have completed execution; and (3) after completion of any outer level command (for example, after execution of a program.)

The outer level of commands is as shown below. (The second line of commands is not displayed on the cathode-ray tube (CRT) screen unless a ? is typed.)

```
-----------------------------------------------------------------------
| Command: E(dit, R(un, F(ile, C(omp, L(ink, X(ecute, D(ebug?         |
|                                                                     |
| ? = Command: A(da, U(ser restart, I(nitialize, H(alt                |
|                                                                     |
-----------------------------------------------------------------------
```

The individual command is executed by typing the character that immediately precedes the (. This character is capitalized in the prompt, but all system command characters may be entered in upper or lower case.

Each outer level command causes execution of a program on the system diskette named "SYSTEM.<function-name>", where <function-name> is, for example, editor, filer, or compiler.

The following paragraphs briefly describe the outer level of commands, which are discussed in more detail in this manual.

- E(dit

  The E(dit command invokes one of the available system editors. The Editors are system programs that allow insertion or deletion of information, finding and replacing character strings, changing text format, copying information, and other text manipulations within a file.

  Entering an E causes the Screen-Oriented Editor to be brought into memory from disk. If the system console is a CRT, the Screen-Oriented Editor is executed. If a work file is present, it is automatically read into the Editor buffer. Otherwise, the Editor prompts for a file.

- R(un

  Entering an R causes the code file associated with the current work file to be executed. If a code file does not currently exist, the system Compiler is called automatically. If the compilation requires linkage to separately compiled code, the Linker also is called automatically and assumes the use of the file *SYSTEM.LIBRARY. The program is executed after a successful compilation and linkage.

- F(ile

  Entering an F calls the File Handler (Filer) into memory from disk. The inner level of commands for the Filer is displayed in a prompt line across the top of the screen after an F is entered.

  The Filer is a system program that provides file maintenance capabilities. For example, the Filer provides facilities for (1) moving, copying, and deleting files; (2) listing volume directories; (3) checking disk or diskette storage for damage or recording errors; (4) naming, or changing the name of, volumes and files; and (5) listing the peripheral devices and volumes currently on line.

- C(omp

  Entering a C initiates the Pascal compiler. If a work file exists the Compiler automatically compiles the work file; otherwise, a prompt for the file to be completed is displayed.

  The Pascal Compiler reads a text file that contains Pascal language statements (source) and converts the statements into executable machine instructions (P-codes).

- L(ink

  Entering an L starts the Linker program which allows routines to be linked from libraries other than *SYSTEM.LIBRARY.

- X(ecute

  Entering an X allows execution of a compiled code file. A prompt asking for the name of the file to be executed is displayed.

  Execution of a program is the actual use of the code file to instruct the computer to do the task for which the program is designed.

If the file requested is present, it is executed. If the file is not present (or the program name is misspelled), the message "No file <file name>.CODE" is displayed.

If the code file is composed of several separately compiled files, one of which has not been linked, the message "Must link first" is displayed.

| NOTE |

> The ".CODE" suffix on a compiled file is implicit and should not be entered as part of the file name.

Programs (particularly programs not yet compiled) can be executed by use of F(ile, G(et the file, Q(uit the Filer, and R(un the file.

The X(ecute command is used to execute the system utilities, like PATCH, SETUP, and so forth. (See Chapter 6 for the details of the system utilities.)

- D(ebug

  Entering a D causes the Debugger utility to be called. If the work file is not compiled or linked, the Compiler and Linker are automatically executed so that a valid code file exists. The Debugger then allows breakpoints to be inserted in the code file and program memory and state to be examined. (See Section 6.15 for details of the Debugger.)

- ?

  The ? is typed to cause the second (and unseen) line of outer level commands to be displayed on the screen.

- A(da

  Entering an A causes the MicroAda(TM) compiler to be called if the compiler is available as part of the operating system. If the compiler is not present, the message "No file:SYSTEM. ADACOMP" is displayed.

  The MicroAda(TM) compiler reads a text file containing Ada language statements (source) and converts those statements into machine-executable instructions.

- U(ser restart

  Entering a U causes the system to begin executing the program or option last used. Using this command is quicker and requires fewer keystrokes than reexecuting the program or reinitiating a specific command.

- I(nitialize

  Entering an I causes the operating system to be reinitialized. That is, the III.Ø Operating System is restarted and the outer level command line is displayed. The assigned volume as the default volume (Filer P(refix command) is maintained across the restart.

  Using the I(nitialize command is not as drastic as using restart button to reinitialize the system.

- H(alt

  Entering an H causes the III.Ø Operating System to terminate; use of this command is not recommended. The system must be reinitialized by using the restart button. The initialization sequence and loading of the system files from disk to memory occur as if the system had just been "powered on".

1.3 INNER LEVEL OF COMMANDS

The inner levels of commands are accessed through the command prompts of the outer level of commands. A brief explanation of the inner level commands is presented in the following subsections. The various commands are explained in more detail in other chapters of this manual.

1.3.1 E(dit

Any one of the four editors may be executed when E is entered from the outer level command line, depending on which editor is named SYSTEM.EDITOR. The Screen-Oriented Editor is named SYSTEM.EDITOR on the operating system disk shipped from the factory, but the any of the other editors could be designated as the system editor. If not renamed, the other editors can be executed by entering X from the outer level command line followed by the file name.

Screen-Oriented Editor

This editor is specifically designed for use with video display terminals (CRTs). This editor provides facilities for manipulating text in the work file or in any text file. The inner level of commands accessed through the Edit command is shown below. (The second line of commands is not displayed on the screen unless a ? is entered from the E(dit prompt line.)

```
-------------------------------------------------------------------------
|>Edit:A(djst C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap ?  |
|                                                                         |
| ? = >move[<arrows>,<sp>,<ret>,=,P(age),direction[<,>],M(rgn,S(et,V(rfy  |
|                                                                         |
-------------------------------------------------------------------------
```

Table 1-1 presents a brief explanation of these commands. (See Chapter 3 for detailed explanations of the commands.)

Table 1-1. E(dit Commands (Page 1 of 2).

Command	Explanation
A(djst	The adjust command allows a line to be shifted left, right, or centered.
C(py	The copy command allows text to be copied from the buffer or a file into the file being edited.
D(lete	The delete command allows text to be removed from the file being edited.
F(ind	The find command allows a specified string of characters to be located in the file being edited.
I(nsrt	The insert command allows characters or spaces to be added to the file being edited.
J(mp	The jump command allows the cursor to be moved quickly through the file being edited to specific points - namely, to the beginning or end of the file and to markers set within the file.
R(place	The replace command allows a specified string of characters in the file being edited to be automatically replaced with a designated string of characters. (Several options regarding the R(place command are explained in Chapter 3.)
Q(uit	The quit command terminates the editing session. Several options regarding the edited file are available when the session is terminated.
X(chng	The exchange command allows a character-for-character change to be effected. That is, a character or space typed over the existing text replaces the existing text with the new character.
Z(ap	The zap command allows sections, lines, words, and so forth of text to be deleted from the file being edited. The text being deleted is stored in the buffer.
?	The ? is typed to cause the second (and unseen) line of E(dit commands to be displayed on the screen.

Table 1-1. E(dit Commands (Page 2 of 2).

Command	Explanation
move [<arrows>,<sp>,<ret>. =,P(age]	This group of actions and the P(age command allow movement through the file being edited.
direction [<,>]	This group of actions allows right, left, up, and down movement through the file being edited.
M(rgn	The margin command is used in conjunction with the S(et command to allow paragraph margins to be specified and automatically adjusted. This command is dependent on the environment being set such that FILLING is true and AUTO-INDENT is false.
S(et	The set command allows the environment to be changed or markers to be set in the file being edited.
V(rfy	The verify command redisplays the text window with the line containing the cursor positioned at the center of the screen.

Advanced Editor (ADV.EDITOR)

The Advanced Editor (ADV.EDITOR) can be renamed SYSTEM.EDITOR and called by entering an E from the outer level command prompt line. Alternately, this editor can be executed by entering an X followed by ADV.EDITOR as the file name. This editor is an enhanced version of the Screen-Oriented Editor. (See Section 3.2.)

L2 Editor (L2)

The L2 Editor (L2), unless renamed SYSTEM.EDITOR, is executed by entering an X from the outer level command prompt line followed by L2 as the file name. The L2 Editor is a version of the Screen-Oriented Editor which allows editing of large files which cannot be contained in main memory at one time. (See Section 3.3.)

Line-Oriented Editor (YALOE)

The Line-Oriented Editor, YALOE, is executed from the outer level command prompt line by entering an X followed by YALOE as the file name. This editor is used when the system console is a teleprinter. This editor, like the Screen-Oriented Editor, provides facilities for inserting, modifying, and deleting text in the work file or any text file. (See Section 3.4.)

1.3.2 File Handler (Filer)

The Filer is the module of the III.Ø Operating System that is used for maintenance of files stored on disk. The Filer is used to view the directory of files, to copy or transfer files between disks, and other file maintenance tasks. The inner level of commands accessed through the F(ile command is shown below. (The second line of commands is not displayed on the screen unless a ? is entered from the F(ile command prompt line.)

```
---------------------------------------------------------------------------
|Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit |
|                                                                         |
|?=Filer: B(ad-blks, E(xt-dir, Krnch, M(ake, P(refix, V(ols, X(amine, Z(ero|
---------------------------------------------------------------------------
```

Table 1-2 summarizes the Filer commands. (See Chapter 4 for detailed explanations of the commands.)

Table 1-2. Filer Commands (Page 1 of 2).

Command	Explanation
G(et	The get command causes the specified file to be loaded into memory from disk.
S(ave	The save command writes the work file to disk as the file name specified in response to the "Save as <file name>?" or "Save as what file?" prompts.
W(hat	The what command displays the name and status (saved or not saved) of the work file.
N(ew	The new command clears the work file space.
L(dir	The list directory command lists the disk directory to the volume and file specified.
R(em	The remove command removes file entries from the directory.
C(hng	The change command changes the name of a file or a volume.
T(rans	The transfer command copies the specified file(s) to a given destination, leaving the source file intact.
D(ate	The date command allows the system date to be changed.
Q(uit	The quit command causes the Filer program to terminate and returns control to the outer level command structure.
?	The ? is typed to cause the second (and unseen) line of Filer commands to be displayed on the screen.
B(ad-blks	The bad-blocks command scans the disk to detect bad blocks (corrupted or damaged storage areas) on the disk.

Table 1-2. Filer Commands (Page 2 of 2).

Command	Explanation
E(xt-dir	The extended-directory command lists the disk directory in more detail than the L(dir command. The additional pieces of information shown by this command are (1) in column one, the unused spaces on the disk; (2) in column two, the beginning block number of the file; (3) in column three, the number of bytes in the last block of the file; and (4) in column four, the file kind.
K(rnch	The crunch command moves the files on the specific volume so that all unused blocks are grouped at the end of the directory (located in the last blocks on the disk).
M(ake	The make command creates a new directory entry with the name specified.
P(refix	The prefix command changes the current default volume to the volume specified.
V(ols	The volumes command lists all the volumes currently on line and off line along with their associated unit (device) numbers to the system console.
X(amine	The examine command attempts to physically recover suspected bad blocks detected by a bad-blocks scan.
Z(ero	The zero command initializes the directory on the specified volume with the new volume name specified and with all blocks on the disk unused.

1.3.3 Other Inner Level Commands

The remaining outer level commands (excluding E(dit and F(ile) access programs that may ask questions, display menus, or display prompt lines for that specific program. For example, the X(ecute command asks for the name of the file to be executed. If, for example, the system utility program PATCH is executed, a series of questions/prompts and command lines are displayed. These various prompts, commands, and menus are also considered inner levels of commands.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

2. SYSTEM FUNDAMENTALS

This chapter describes the files and volumes (I/O devices) allowed with the III.Ø Operating System. Basic knowledge of the types of files and file specifications is essential to effective use of the operating system. Likewise, some basic information regarding the use of volumes is necessary to be able to take advantage of the features available in the III.Ø Operating System.

This chapter provides those essential facts and concepts that enable effective programming and ease of use of the operating system.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

2.1 FILES

A file is defined as a body of information or a stream of bytes that is usually stored on an I/O device. Typical examples of files are programs, letters, lists, and text stored on disks or diskettes as well as information sent to a printer. For diskettes or disks, a table of contents for the files stored on the disk or diskette is maintained. This table of contents is called a directory; each file has a separate entry in the directory. Any file is referenced according to the file name by a Pascal program and by the III.Ø Operating System. Each entry in the directory is a file name.

The directory shows certain pieces of information regarding the file. The file name that is given to the file plus the type of file are two of those pieces of information. The most common files are either text or code files. Text files contain information such as letters, lists, reports, and program source statements. Code files contain the P-codes (machine-executable information) for a source program. The file type or kind is denoted by the suffix appended to the file name. The following types of permanent files are used by the III.Ø Operating System; one additional file is used by the III.Ø Operating System - a work file. (This file is discussed separately in a following subsection.)

Reserved Suffix	Contents of File	Extended Directory Listing
.TEXT	Human-readable text	Textfile
.CODE	Machine-executable code	Codefile
.DATA	Data file	Datafile
.BAD	A physically damaged area of disk	Bad file

These file types are explained more fully in the following subsections.

The directory of any given volume is limited to 77 file entries. If the directory is full and an attempt to write a new file to that volume is made, the following error message results:

```
| No room on disk                                                      |
```

The above error message also results when an area on disk does not exist that is large enough to contain the file.

2.1.1 Text Files

A text file contains human-readable text. The text file is composed of 1024 byte pages, where a page is defined as:

<[DLE][indent][text][CR][DLE][indent][text][CR]. . .[nulls]>

Data Link Escapes (DLEs) are followed by an indent code, which is a byte that contains a value 32 plus the number of spaces for indentation. At the end of the page, the last carriage return is followed by at least one null. The nulls pad to the end of the page to give the Compiler an integral numbers of lines on a page. The DLE and indent code are optional and are used for text compression.

The first page of a text file is the header page. This page is reserved for information for the Text Editor. When a user program opens a text file and REWRITEs or RESETs it with a file name ending in .TEXT the I/O subsystem creates, then skips, the header page. The Filer transfers the header page only on a disk-to disk transfer; the header page is omitted on a transfer to a serial device (PRINTER or CONSOLE).

2.1.2 Code Files

A code file is the file generated by compiling a program. A program is generally contained in a text file (the source statements written in the programming language) which is compiled; on successful completion of the compilation, a code file is generated. This file contains machine-executable instructions (P-codes) that were generated from the source program. The suffix .CODE is automatically appended to the original file name to designate the code file that matches the text file.

The first block of information in a code file describes the code kept in the file. Heading the block is an array of 16 word pairs - a pair for each segment on the disk. (With the H2 release, information for the additional segments (128 segments available) is stored in segment pages at the end of the file.) The first word of the pair gives the block number within the file where code begins. The second word gives the number of words of code located there.

Following this array is a series of 16 eight-character arrays that describe the segments by name. These eight characters identify the segment at compile time.

Then follows a 16-word array of state descriptors. The values in this array tell what kind of segment is at the described location. The values are:

LINKED
HOSTSEG
SEGPROC
UNITSESG
SEPRTSEG

The remaining 144 words of the block are reserved for system use.

2.1.3 Data Files

The content and format of data files are determined by the user.

2.1.4 Bad Files

Bad files are those files marked by the Filer after a bad-block scan detects bad blocks, and the bad blocks have been examined. (See section 4.2.11.) The designation of bad files prevents use of physically bad blocks on disk.

2.1.5 Work File

A file basic to the III.Ø Operating System is the work file. The work file concept is that space is temporarily available for a copy of a file being created or one being changed. This space and a name are reserved for any work that is being done on the system. If a specific name is not assigned to a text file at the completion of the work session and the Update option of the Editor is selected to end the session, the III.Ø Operating System automatically assigns the name *SYSTEM.WRK.TEXT to the file and then writes the file on the system diskette.

If the work file is a Pascal program, the R(un command can be used to compile and then execute the code. The R(un command causes the Pascal compiler (1) to take the current work file; (2) compile it into executable P-code; and (3) when no errors are found and compilation is completed to call the III.Ø Operating System to execute the code. The Pascal compiler saves the code form of the work file on the system diskette as *SYSTEM.WRK.CODE.

Thus, the work file can be edited, compiled, linked, or run numerous times without telling the III.Ø Operating System that the file to use is the work file. Each of the above operations is designed to use the work file on the operating system diskette unless a specific file name is entered.

Thus, a program can be written and debugged with a minimum amount of keystrokes and without redundant write operations. Once the program is completed and runs correctly, the text and code work files can be given permanent names so that the program is stored on disk. The work files are not permanently saved on disk until the Filer S(ave command is executed, and the work files named. Once the new name is entered in response to the S(ave prompt "Save as what file?", both work files are renamed and written onto the disk.

2.1.6 File Names

Because Pascal programs and the III.Ø Operating System reference a file by its name, a correct file name is important. The following rules and statements define a legal file name.

- The file name may not exceed 15 characters. (The volume name may be specified in addition to the 15 characters. However, the volume prefix may not exceed seven characters plus the colon.)

- The file name may not include the following characters: "=", "$", "?", or ",".

- The legal characters for a file name are the alphanumerics plus the following special characters: "-", "/", "\", " ", and ".".

- Lower-case letters used in a file name are translated to upper case.

- Blanks and nonprinting characters used in a file name are removed.

Special characters are normally used to indicate hierarchical relationships between files and to distinguish related files of different types.

The wild card characters "=" and "?" are used to specify subsets of the directory. (See Section 4.1.1). Many Filer commands use a file specification to perform a certain action on the group of files designated.

2.2 VOLUMES

A volume is any I/O device (that is, a device connected to the computer to send or receive data.) A block-structured device is one that can have a directory (for example, disk). A non-block-structured device does not have an internal structure; it simply produces or consumes a stream of characters (for example, printer and console). A non-block-structured device can be referenced by the device file name (such as PRINTER: or CONSOLE:) or by the unit number. Block-structured devices can be referenced by the unit number or by the volume name of the diskette stored in the appropriate drive or the volume name as configured on a Winchester disk.

Table 2-1 gives the volume names reserved for non-block-structured devices, the unit number associated with each device, and the unit numbers associated with the system and alternate disks.

Table 2-1. I/O Devices.

Unit Number	Volume ID	Description
1	CONSOLE:	Screen and keyboard with echo
2	SYSTEM:	Screen and keyboard without echo
3		UNUSED
4	<volume name>:	System disk (typically)
5	<volume name>:	Alternate disk (Winchester or floppy)
6	PRINTER:	Line printer (parallel device)
7	RCONS1:	Remote console
8	REMOTE:	Additional peripherals (serial devices)
9-14	<volume name>:	Additional disk drives (Winchester or floppy
15	RCONS2:	Remote console
16	RTERM2:	Remote terminal
17	RCONS3:	Remote console
18	RTERM3:	Remote terminal
19	RCONS4:	Remote console
20	RTERM4:	Remote terminal
21	RCONS5:	Remote console
22	RTERM5:	Remote terminal
23	RCONS6:	Remote console
24	RTERM6:	Remote terminal
25	RCONS7:	Remote console
26	RTERM7:	Remote terminal
27	PRINTR1:	Additional line printer
28..255		Winchester disk units or future devices

(Additional unit numbers are reserved for system use.)

On H3 and later releases, the unit numbers 4, 5, 9-14, and 28-255 can be used as either Winchester or floppy units. In general, any unit can be the "system" unit, depending on how the system is configured. The default configuration for a floppy-only system is shown in Table 2-1.

The volumes CONSOLE: AND SYSTERM: refer to the user CRT and keyboard. In a Pascal program, CONSOLE: is referenced by the standard file names INPUT and OUTPUT; SYSTERM: is referenced by the standard file name KEYBOARD:. The difference between SYSTERM: AND CONSOLE: is that reading from CONSOLE: causes input characters to be echoed to the screen and reading from SYSTERM: does not. This difference in character echo also applies to RCONS: and RTERM:.

Volume names for block-structured devices can be assigned by the user. The following rules and statements define a legal volume name.

- The volume name may not exceed seven characters in length.
- The volume name may not contain the following characters: "=", "$", "?", or ",".
- The character "*" is the reserved volume ID of the system disk, the disk on which the system was booted.

The character ":" when used alone is the volume ID of the default disk. The system and default disks are equivalent unless the default prefix is changed using the Filer P(refix command.

Use of the "#<unit number>" is equivalent to the name of the volume in the disk drive at the current time or designates another I/O device (for example, #6: designates the PRINTER:).

3. SYSTEM EDITORS

An editor is a specialized program that facilitates creating, reading and changing text files. The III.Ø Operating System contains four editors: the Screen-Oriented Editor (SYSTEM.EDITOR), the Advanced Editor (ADV.EDITOR), the L2 Editor (L2), and a Line-Oriented Editor (YALOE). Each of these editors is suited to a specific use. That is, the Screen-Oriented Editor is designed for use with a video display console; it handles a text file as one unit in the main memory of the computer. The Advanced Editor is an enhanced version of the Screen-Oriented Editor designed to offer additional efficiencies in text manipulation. The Advanced Editor is included in the III.Ø Operating System as an alternate choice to the Screen-Oriented Editor as the system editor. The editor named SYSTEM.EDITOR is invoked when an E is typed at the outer command level. The L2 Editor is a version of the Screen-Oriented Editor which facilitates editing of large files which cannot fit into the main memory buffer at one time. The Line-Oriented-Editor (YALOE) is designed for use with a teleprinter or telewriter as the system console.

These four editors plus their commands are described in this chapter.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

3.1 SCREEN-ORIENTED EDITOR

The Screen-Oriented Editor is designed for use with video display terminals. This editor handles a text file as one unit which is read into the main memory buffer of the computer. The Screen-Oriented Editor facilitates text manipulation by providing such capabilities as insertion and deletion of text, change of text character-for-character, setting and modifying paragraph margins, finding a specific character string, moving text from one place to another, and replacing a given character string with another.

3.1.1 General Information

The Screen-Oriented Editor provides a window into the file through the screen of a CRT. The window shows that portion of the file in which editing is taking place. The window can be moved to various parts of the file displaying the portion of the text available at that position.

When entering any file, the Screen-Oriented Editor displays the start of the file in the upper left corner of the screen. That position is the original position of the cursor. The cursor is a marker indicating the position at which an action would take place if initiated. The cursor can be moved about freely in the file by the directional arrows until an editing command or mode is specified. Once the command is executed, the cursor is frozen within the movement specifications of the command and cannot be freely moved until that action is completed.

The cursor is never actually "at" a character position but is between the character at which it appears (for ease of display) and the character immediately preceding. This location is most clearly apparent in the I(nsert mode, which inserts in front of the character at which the cursor is located.

Repeat factors are allowed by many of the commands to repeat the action of the command as many times as indicated by the immediately preceding number. For example, entering 2 <down-arrow> causes the <down arrow> command to be repeated twice, moving the cursor down two lines. The assumed repeat factor is 1 if no number is typed before the command. A slash (/) typed before the command indicates an infinite number of repeats for some commands.

Some commands are directional. If their direction is forward, they operate forward through the file; if backwards, they operate in reverse. The directional arrow that appears before the "Edit:" command line indicates, for example, the default direction for commands that are directional. The right arrow (>,"greater than" sign) appears at the beginning of the "Edit:" command line. Unless the dirction is changed, this arrow indicates that all directional actions will progress forward through the file. When direction affects the commands, it is specifically noted in this manual.

All command characters may be entered in upper or lower case, although they are referenced in this document in upper case form only for brevity.

3.1.2 Accessing the Screen-Oriented Editor

The Screen-Oriented Editor is accessed by typing E (for edit) from the outer level command prompt line. If a work file exists, this editor automatically reads it into the main memory buffer for editing. If a work file does not exist, the following prompt appears on the screen:

```
>Edit:
 No workfile is present. File? (<ret>for no file  <esc-ret> to exit)
 :
```

If a return (<ret>) is entered, the Edit command line appears across the top of the screen. The main Edit command line is illustrated below.

```
>Edit:A(djst C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap?
```

The second Edit command line can be accessed by typing a ?. The second Edit command line is illustrated below.

```
>move[<arrows>, <sp>,<ret>,=,P(age], direction [<,>],M(rgn,S(et,V(rfy
```

If a file name for editing is entered in response to the first prompt, the following lines appear on the screen as the file is read into the buffer; then, the Edit command line appears across the top of the screen.

```
--------------------------------------------------------------------------
|>Edit:                                                                  |
| Reading....                                                            |
--------------------------------------------------------------------------
```

If a file name is entered that is not present (for example, a typographical error is made in the file name), the following message and prompt are displayed.

```
--------------------------------------------------------------------------
| Not present. File?                                                     |
| :                                                                      |
--------------------------------------------------------------------------
```

Once the file is read into the buffer or a new file is designated, the cursor is shown in the upper left corner of the screen. Unless the first line is indented, this position is row 1 column Ø of the screen and is the beginning of the file.

3.1.3 Screen-Oriented Editor Commands

Although the Screen-Oriented Editor commands are described in this manual in the order in which they appear in the Edit command prompt line, the commands can be grouped into three major categories, as follows:

- Moving commands.
- Text-changing commands.
- Formatting commands.
- Quit command.

A brief discussion of each of these categories follows.

- MOVING COMMANDS

  The moving commands move the cursor from one location to another to position it for the next editing function. Many of these commands are initiated by keys on the CRT keyboard. The commands initiated from the CRT keyboard are listed in Table 3-1.

Table 3-1. Moving Commands - CRT Initiated.

Command/Key	Function
<down-arrow>	Moves cursor down
<up-arrow>	Moves cursor up
<right-arrow>	Moves cursor right
<left-arrow>	Moves cursor left
"<" or "," or "-"	Changes the direction to backward
">" or "." or "+"	Changes the direction to forward
<space>	Moves 1 character (directional)
<backspace>	Moves cursor left
<return>	Moves to the beginning of the next line (directional)
P(age	Moves the screen display on screen page forward or backward (directional)
J(ump	Moves the cursor to a predetermined point in the file as follows:
B(egin	Moves cursor to the beginning of the file
E(nd	Moves cursor to the end of the file
M(arker	Moves cursor to the marker specified

Direction is always indicated by an arrow (> or <) in front of the prompt line. The direction is forward when the Editor is entered, but can be changed by typing the appropriate arrow whenever the "Edit:" prompt line is present. On many standard keyboards, the period (.) can be used for forward because it is the lower case for ">"; and the comma (,) can be used for backward, being the lower case for "<". Also, the + and - signs change the direction -- + is forward and - is backward.

Repeat-factors are valid for some command options and some of the cursor moves. A repeat-factor is a number that specifies how many times the command function or move action is to be repeated. The number is entered immediately prior to the cursor move or command option. For example, the F(ind and R(eplace commands allow repeat-factors. Also, use of the down or up arrows allows a repeat-factor to be specified.

The cursor moves and other commands that allow repeat-factors use a factor of 1 if no number is specified. Repeat-factors may range from Ø to 9999 when entered as a number. Using the slash (/) before a cursor move causes the action to repeat infinitely until the end of the file (or beginning of the file, depending on the direction) is reached. Using the slash (/) with other commands that allow repeat factors causes the last occurrence of a string in the file to be found or an infinite repeat of the command. For example, if "/RLV.pascal ..Pascal." is entered from the Edit command line, all occurrences of "pascal" in the file are found on a one-by-one basis, the cursor appears at the end of each target, and a prompt appears for a decision as to whether or not to replace that occurrence with the substitute string. (See the Replace command subsection for additional explanation of these actions.)

Repeat factors can be used with any of the keyboard commands listed in Table 3-1. Repeat factors are ignored if not appropriate to the command (such as "<" or ">" direction changes).

The Editor maintains the column position of the cursor when executing the <up arrow> and <down arrow> commands.

The moving commands that do not have special function keys on the CRT keyboard are JUMP, PAGE, and = (equals); these commands are described in separate subsections.

- TEXT-CHANGING COMMANDS

The majority of Editor commands fall into the text-changing category. The main function of an editor is to facilitate the manipulation of text within a file. The text-changing commands are listed below but are described in separate subsections.

C(py	(Copy Command)
D(elete	(Delete Command)
I(nsrt	(Insert Command)
R(place)	(Replace Command)
X(chng	(Exchange Command)
Z(ap	(Zap Command)

- FORMATTING COMMANDS

Several Edit commands effect text formatting. This group of commands control indentation, margins, and general text layout on the page. These commands are listed on the following page but are described in separate subsections.

```
A(djst        (Adjust Commands)
M(rgn         (Margin Commands)
S(et          (Set Commands)
```

A(djst (Adjust Command)

The Adjust command allows selected lines of text to be shifted right or left without changing their contents. This command is initiated by typing an A from the Edit command prompt line. After entering the A, the following prompt line appears:

```
 -----------------------------------------------------------------------
 |>Adjust: L(just R(just C(enter <left,right,up,down-arrows>{<etx> to leave}|
 -----------------------------------------------------------------------
```

These options refer to the line on which the cursor is located. This command adjusts indentation on a line-by-line basis. On any line, the right-arrow and left-arrow commands move the whole line one space to the right or left, respectively, each time the arrow is typed. An <etx> or (editor accept key) is typed when indentation is adjusted as desired.

To adjust a sequence of lines, one line is adjusted; then the up-arrow and down-arrrow commands are used to adjust the line above or below, respectively, by the same amount. Repeat factors can be used before any of the arrows; use of the / is also valid.

"L" and "R" are used to left- and right-justify lines to margins set in the Environment. "C" centers the line between the set margins. Typing an up- or down-arrow justifies or centers the line above or below to the same specification as the original line.

The Adjust command can only be terminated by typing an <etx> (or equivalent); an adjust action can be aborted by typing <esc> before any line adjustment is specified.

C(py (Copy Comand)

The Copy command allows insertion of passages of text into the work file; the insertion may be text previously saved in the buffer of the work file or text copied from a file other than the work file. This command is initiated by typing a C from the Edit command line. After entering C, the following prompt line appears:

```
 -----------------------------------------------------------------------
 |>Copy: B(uffer  F(from file  <esc>                                    |
 -----------------------------------------------------------------------
```

C(PY B(UFFER

The C(py B(uffer option copies the text saved in the buffer into the work file at the cursor position where the C was entered. Each use of an I(nsrt, D(lete, or Z(ap command stores the text passage that was inserted, deleted, or zapped, in the buffer. Thus, through use of D(lete, then terminating the deletion with an <esc> instead of an <etx>, the C(py B(uffer option allows the text to be copied at a second location in the file but leaves the original text intact. That is, the sequence -- D(elete <esc> C(opy B(uffer -- allows copying text; the sequence -- D(elete <etx> C(opy B(uffer -- allows moving text. Any insertion or deletion of text before copying the buffer automatically fills the buffer with that text and, in so doing, removes the text previously stored in the buffer.

Figure 3-1 is an example of the C(py B(uffer selection using a "D(lete <esc>" sequence first in order to copy a passage of text to a second location in the file. In Figure 3-1, the keys typed are shaded; comments are enclosed in braces ({}). The Edit command line and the text passage to be copied are shown at the top of the figure. The cursor is located at the beginning of the text to be deleted/copied.

After the copy is completed, the cursor returns to the position immediately preceding the text that was copied. The use of the C(py B(uffer sequence does not change the contents of the buffer. The original indentation of complete lines in the buffer is retained when the buffer is copied into the file.

C(PY F(ILE

The C(py F(ile option is used to copy another file or a passage of text from another file into the work file. To copy a passage of text from another file that is saved on disk, markers must have been previously set while editing that file. The text to be copied must be delimited by a beginning and ending marker.

When the C(py F(ile option is selected by first entering C for copy from the Edit command line and then entering an F for "F(rom file" from the Copy prompt line, the following prompt appears. The file name and the appropriate marker names are requested.

```
---------------------------------------------------------------------
|>Copy: From what file[marker,marker]?                              |
---------------------------------------------------------------------
```

Any file may be specified; however, a text file is assumed. The copy operation does not change the contents of file being copied. Also, the original indentation of complete lines in the external file is retained when the file (or portion of it) is copied into the file being edited.

The sequence of entries and responses in Figure 3-2 illustrates the use of the C(py F(ile option to copy a portion of text (delimited by markers a and b) from a second file named "cpyex.text" into the current work file. In Figure 3-2, the keys typed to effect the copy are shaded; comments are enclosed in braces ({}). System responses are prefaced by a right arrow (>).

```
----------------------------------------------------------------------
>Edit: A(djst C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chnge Z(ap ?
 D <ret>  Fill in the following information
   <ret>  to allow an update of your
   <ret>  credit record.
   <ret>  Name:___________________________
   <ret>  Address:________________________
   <ret>  Acct. Number_____________________
   <esc>                                                          (1)
                    .
                    . {Continuation of file text}
                    .

          Your credit record can be an important
          asset when applying for a loan.  Your
          rights to have knowledge of your credit
          rating have been established by law.
          To find out your credit rating, return
          the portion of this flyer marked below.

          ---------------------------------------
   C        <---{Cursor is positioned on the blank line below the line
                 of hypens in the text.}
   B                                                              (2)
          Fill in the following information
          to allow an update of your
          credit record.
          Name:___________________________
          Address:________________________
          Acct. Number_____________________

          ---------------------------------------

          If you have questions regarding the
          procedure explained in this flyer, call
          8ØØ-222-1ØØØ. {End of file}
---------------------------------------------------------
| (1) The "D(lete <esc>" sequence causes the text to dis-|
|     appear from the screen - then to reappear after the|
|     buffer is filled.                                  |
|                                                        |
| (2) The "COPY B(uffer" sequence is a two-step action   |
|     that causes a COPY menu to appear then disappear as|
|     the selection is made.  The copy is then completed.|
---------------------------------------------------------
```

Figure 3-1. C(py B(uffer Example.

```
>Edit: A(djst C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap ?

    C
    >Copy: B(uffer F(rom file <esc>
    F
    >Copy: From what file[marker,marker]? #5:cpyex.text[a,b]<ret>
    >Copy ...............

    {After the above return is typed, the copy is completed, beginning
     at the location of the cursor when the C for copy was entered.}
```

Figure 3-2. C(py F(ile Example - Passage of Text.

If no marker names are entered with the file name, the entire file designated is copied into the work file. The copy begins at the cursor location where the copy operation was begun. On completion of the copy from a file, the cursor returns to the beginning of the text just copied from the file.

D(lete (Delete Command)

The Delete command is initiated by typing D for delete from the Edit command line. This command allows characters to be removed from the text being edited. After typing a D for delete, the following prompt line appears across the top of the screen.

```
|>Delete: <> <Moving commands>{<etx> to delete,<esc> to abort}      |
```

To delete characters, any of the cursor moving commands (<arrows>, <ret>, and so forth) are valid. The arrow before the word "Delete" in the prompt line indicates the direction in which the characters are to be deleted. The direction can be changed by typing the directional arrow just prior to typing the D for delete or during the delete action.

Typing <ret> while in Delete mode removes the entire line of text. Also, the repeat factor may be used to delete several lines at once by prefacing a <ret> (or any other moving command) with the desired repeat number.

The cursor must be placed at the first character to be deleted. This position is the anchor position or starting point. As the cursor is moved away from the anchor position, text in its path is removed. As the cursor is moved back toward the anchor position, previously deleted text is restored. All text between the anchor position and the final position is deleted, and the space is closed up when the <etx> (or editor accept key) is typed.

The Delete command is terminated in one of two ways - (1) either typing an <etx> to accept the deletion or (2) typing an <esc> to abort the deletion. Typing an <esc> leaves the original text in place in the file. For either termination, the text that is or would have been deleted is copied into the buffer. (Refer to the C(py Command section in this chapter.)

Figure 3-3 illustrates use of the Delete command. The keys typed to effect the deletion are shaded; comments are enclosed in braces ({}). System prompts are prefaced by a directional arrow.

```
>Edit:A(djst C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap ?

  {The text below is the original text before deleting.}
  This sentence of the text is to remain the same.  This sentence
  is to be modified by the delete operation.

  {Position the cursor over the "t" in the second occurrence of "to."}
  D,<space>,<space>,<space>,<space>,<space>,<etx>

  {The following text results from the deletion.}
  This sentence of the text is to remain the same.  This sentence
  is modified by the delete operation.
```

Figure 3-3. Example of the Delete Command.

A padded ' ' (space) may be implicitly added by the operating system to the end of the line following the deletion.

After a deletion that includes a <ret>, the line on which the cursor is located may extend beyond the edge of the screen display (80 characters). An ! appears in the last visible character position of the line to indicate that text occurs beyond the screen limit and cannot be displayed. To see the text that extends beyond the screen limit, a <ret> can be inserted anywhere in the visible portion of the line. The text that was not seen is then displayed on a new line below the visible text. The new line of text begins with the character on which the cursor was located when the <ret> was inserted.

F(ind (Find Command)

The Find command searches through the file for the specified group of characters (the target) and moves the cursor to the end of that group. If a repeat-factor is specified, the Find command moves the cursor to the end of the specified occurrence of the target.

The Find command is initiated by typing an F from the Edit command line. After an F is entered, one of the following two prompt lines is displayed depending on the setting of the T(oken default option in the Environment mode. (See the S(et E(nvironment description in a separate section of this chapter.)

```
|>Find [1]:L(it<target>=>                                               |
```

The above prompt line appears if the T(oken default is set to true.

```
|>Find [1]:T(ok<target>=>                                               |
```

The above prompt line appears if the T(oken default is set to false.

TARGETS AND DELIMITERS

The target is a group of characters and/or spaces that is specified as the string (or group) to be found. The target is described or "set off" to the system by delimiters. Delimiters are a set of characters that enclose the specified target when entering the command. Any character that is not a letter or number may be chosen as the delimiter as long as the character is not in the target string.

If a character that occurs in the target is used as a delimiter, the Find action begins immediately after the character is entered. The Editor interprets that character as the closing delimiter and thus, begins searching for the target string. In that case, the target found is only part of the intended target.

A commonly used delimiter is the slash (/) because that character does not often occur in text, and it is convenient to type.

If the target is not preceded by a delimiter, the following error message appears:

```
--------------------------------------------------------------------
|ERROR: Invalid delimiter. <space> to continue.                    |
--------------------------------------------------------------------
```

For a literal search, the Find command searches for the target string exactly as it is entered. That is, if the target is entered in all capital letters, the search is for a matching pattern in the file - all capital letters. If, for example, the target is entered in all caps and the pattern in the file only begins with a capital letter, a match is not made and the following message appears.

```
--------------------------------------------------------------------
|ERROR: Pattern not in the file. <space>to continue.               |
--------------------------------------------------------------------
```

LITERAL AND TOKEN SEARCHES

The search for the target may be either a literal or token search; the target is treated differently for each of these options. The literal search causes the target to be matched exactly or literally, even if the target appears within a word. The spaces are also considered in a literal search. For example, the literal target / Pascal / produces only the match of " Pascal " (as a separate word in the file). However, the word Pascal followed by a period (Pascal.) would not match the target because the target is enclosed by spaces. Also, a literal target like /oper/, might match the following patterns in text:

```
operating
operation
operate
cooperate
```

The token search matches the target to a token, which may be a complete word, a punctuation character, or an identifier. Several different tokens may be strung together to form a single target. Blanks or spaces are not considered in the token search. For example, a target of /I:INTEGER;/ when used in a token search could match the following patterns in the work file:

```
I:INTEGER;
I: INTEGER;
I : INTEGER;
I:    INTEGER;
I:
INTEGER;"
```

The default (or automatic) setting for either literal or token searching is determined by the setting of the T(oken default option, accessed by the S(et E(nvironment command. The Find prompt line displays the alternate search type - either L(it or T(ok - not the default type. That is, the default type is the one NOT shown in the prompt line.

To use the alternate type rather than the default type, the first letter (as shown in the prompt line) of the alternate search is typed before the target is entered but after the F for Find is entered. The letter appears after the => of the prompt line; no action begins until the closing delimiter of the target is entered.

SAME-TARGET OPTION

In order to find repeated occurrences of a target in a file, typing FS causes the Editor to search for the target string last specified. Thus, the target need not be reentered. However, L(it or T(oken is a property of each find action; this property is not associated with the pattern when it is defined. For example, the sequence -- FL/oper/ -- finds the next occurrence of "oper", but an "FS" following that sequence does not find the next occurrence of the target because the L (for literal) must be typed again.

If the last specified target is not known, the S(et E(nvironment command can be executed to show the current target. For example, the Environment display might list the following:

```
Patterns:
<target>= 'Pascal'
```

I(nsrt (Insert Command

The Insert command allows new information to be added to the work file. All characters typed as an insertion become part of the text stored in main memory. To insert text, the cursor is positioned at the place the insert is to begin. An I for insert is typed from the Edit command line; the following prompt line appears:

```
-------------------------------------------------------------------------------
|>Insert: Text {<bs> a char,<del> a line} [<etx> accepts, <esc> escapes]  |
-------------------------------------------------------------------------------
```

In the insertion, the new characters are added between the character on which the cursor was located when the insert began and the character to the immediate left of the cursor. That is, a space is opened between the cursor position and the character to the left of the cursor. This space continues to widen as characters are entered; the original text that was to the right of the cursor is moved right as the insert increases. The shifting of text continues until the insert is finished and accepted (<etx> or editor accept key) or until the insert is aborted (<esc>).

Once the original text is pushed to the screen display limit, that line drops down to the next line to allow more text to be inserted. Once the insertion reaches the screen display limit, the original text that made up the remainder of the line drops down another line. Therefore, when a <ret> is typed, the insertion can continue on a new line. The rest of the work file page of text is not displayed on the screen but remains in main memory. When an <etx> is typed to accept the insertion, the original text is brought to the end of the insertion and the remainder of the page appears on the screen.

If a <ret> is inserted at the screen display limit, the original text to the right of the insert disappears from the screen, allowing as many new lines as required to be inserted. After a <ret> is inserted, the cursor is positioned immediately below the first character of the line above, if A(uto indent is true. If A(uto indent is false, the cursor is positioned to column Ø or the left margin. To change the indentation of the new line, the space bar or backspace key can be used to alter the cursor position. This alteration must be done immediately after the <ret> is typed and before any text is entered. Once any character other than space or backspace is typed at the beginning of the line, the indentation cannot be altered by the space or backspace keys.

CORRECTING ERRORS

The Insert prompt line shows the error-correcting capabilities available during the insert. The <bs> corresponds to the left-arrow key (backspace) and is used to delete a character at a time in the reverse direction. The <del> corresponds to the delete key, which deletes all text back to and including the last <ret> character entered.

| NOTE |

> The direction set at the beginning of the Insert prompt line is not valid. If a nonusable control character like an up arrow, is typed inadvertently, a question mark (?) appears on the screen. These errors can be erased by the <bs> or <del> keys.

ACCEPTING OR ABORTING THE INSERTION

To end the insertion (accept the new text into the file), an <etx> (or editor accept key) is typed. To abort the insertion at any point, an <esc> is entered. All inserted text is discarded when the <esc> key is typed. That is, the copy buffer is not changed by an I(nsert <esc>.

However, after an insertion is accepted (<etx>), the information is available from the copy buffer until the next insertion or deletion. Therefore, if an insert is to appear in several locations in the file, the C(opy B(uffer command can be used to place the text in the various location.

J(mp (Jump Command)

The Jump command allows the cursor to be moved quickly from one place to another in the file without using the up or down arrows repeatedly. The Jump command moves the cursor to the beginning or end of the file or to preset markers in the file.

The Jump command is initiated by typing a J from the Edit command line. The following prompt line appears:

```
--------------------------------------------------------------------------
|>Jump: B(eginning E(nd Marker <esc>                                      |
--------------------------------------------------------------------------
```

Typing a B for beginning moves the cursor to the beginning of the file, displays the Edit command line at the top of the screen, and displays the first page of the file. Likewise, typing an E for end moves the cursor to the end of the file, displays the Edit command line at the top of the screen, and displays the last page of the file.

Typing an M for Marker causes the following prompt line to appear.

```
--------------------------------------------------------------------------
|Jump to what marker?                                                     |
--------------------------------------------------------------------------
```

If a marker name is entered that is present in the file, the cursor moves to that position after a <ret> is typed. If a nonexistent marker name is entered, the following error message appears:

```
--------------------------------------------------------------------------
|ERROR: Not there. <space> to continue.                                   |
--------------------------------------------------------------------------
```

The cursor does not move from its current position when an error occurs. Establishing markers in the file is explained in the S(et M(arker command section.

If <esc> is typed in response to the jump prompt line, the jump action is aborted.

R(place (Replace Command)

The Replace command finds a target and replaces it with a specified substitute. This command is very similar to the Find command but extends the capabilities of Find. (See the section discussing the Find command.)

The Replace command is initiated by typing an R from the Edit command line. After an R is entered, one of the following two prompt lines is displayed depending on the setting of the T(oken default option in the Environment mode. (See the S(et E(nvironment description in a separate section in this chapter.)

```
|>Replace [1]:L(it V(fy <targ><sub> =>
```

The above prompt line appears if the T(oken default is set to true.

```
|>Replace [1]:T(ok V(fy <targ><sub> =>
```

The above prompt line appears if the T(oken default is false.

The Replace command searches through the file according to the direction set, finds the specified number of occurrences of the target, and replaces each occurrence with the specified substitute (unless verification is selected.) After the replacement is completed, the cursor is positioned at the end of the last target found or substituted.

See the Find command section for a discussion of the repeat-factor, targets and delimiters, and the literal and token search modes.

COMMAND STRUCTURE

The Replace command requires two user-specified groups of characters - the target (same as the Find command) and the substitute. The target is the group of characters to be found, and the substitute is the new replacement for the target.

These strings must each be enclosed within a set of delimiters. Delimiters must form a set; that is, the opening and closing delimiter must be the same character.

A typical example of the Replace command structure is given below:

```
--------------------------------------------------------------------------------
|>Replace[1]:L(it V(fy <targ><sub>=>/ pascal// Pascal/                          |
--------------------------------------------------------------------------------
```

The slashes are the delimiters. The replace operation would replace the first occurrence of the token " pascal" with " Pascal", starting at the cursor position and replacing forward in the file.

VERIFY OPTION

When V is entered (for verification) in the Replace command, no substitute of characters is completed until the user looks at each target found and decides to replace that occurrence. After the V is typed in the command, no action occurs until the first occurrence of the target is found. At that point, the following prompt asks the user for a decision regarding the replacement.

```
--------------------------------------------------------------------------------
|>Replace[1]: <esc> aborts,'R' replaces,' ' doesn't                             |
--------------------------------------------------------------------------------
```

If the user wants to replace the target with the substitute, an R is typed. If the user does not want to replace that occurrence of the target with the substitute, a <space> is typed. To abort the replace operation, an <esc> can be entered.

A slash (/) used with the Verify option causes every occurrence (in the set direction) of the target to be examined before replacement.

SAME-STRING OPTION

As with the Find command, the same-string option is available with the Replace command. Typing an S in place of the target directs the Replace command to use the target specified previously, either by a previous use of the Replace or Find commands. Likewise, an S may be used for the substitute string. The Replace command then uses the last substitute string specified in a previous Replace command.

For example, the following Replace command entry causes the command to use a previous target with a new substitute string.

```
|>Replace: L(it V(fy <targ><sub>=>S/Pascal(TM)/                        |
```

Likewise, the following Replace command entry causes the command to use a previous substitute with a new target.

```
|>Replace: L(it V(fy <targ><sub>=>/pascal/s                            |
```

Typing the following characters causes the Replace command to use the previous target and substitute:

RVSS

The next occurrence of the previously specified target is replaced (after verification) with the previously specified substitute.

If a previous target or substitute has not been specified the following message appears:

```
|ERROR: No old pattern. <space> to continue.                           |
```

Figure 3-4 is an example of use of the Replace command. In Figure 3-4 user input is shaded; comments are enclosed in braces ({}). System responses are prefaced by an >.

```
>Edit:A(djst C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap ?

  PROGRAM REPLACE; {Text before replacement.}
    BEGIN
      WRITELN ('SOME WORDS');
      WRITELN ('MORE WORDS');
      WRITELN ('EVEN MORE WORDS');
    END.

3R
>Replace [3]:L(it V(fy <targ><sub>=>/WORDS//BYTES/

  PROGRAM REPLACE; {Text after replacement}
    BEGIN
      WRITELN ('SOME BYTES');
      WRITELN ('MORE BYTES');
      WRITELN ('EVEN MORE BYTES');
    END.
```

Figure 3-4. Example of the Replace Command.

Q(uit (Quit Command)

The Quit command terminates the Editor session. The Quit command is initiated by typing Q from the Edit command line. The following message appears:

```
|>Quit:
|   U(pdate the workfile and leave
|   E(xit without updating
|   R(eturn to the editor without updating
|   W(rite to a file name and return
|   S(ave as <vol:file name> and return
```

One of the five options must be selected by typing the appropriate letter. These five options are described in the following subsections.

UPDATE OPTION

The U(pdate option causes the editor to write the file just modified (currently in memory) onto the system volume as SYSTEM.WRK.TEXT. This option erases any previous versions of the system work file. (SYSTEM.WRK.CODE is removed as well as the previous SYSTEM.WRK.TEXT.)

If the system work file is the text file being edited, the U(pdate option should be used periodically to avoid accidental loss of recent changes.

EXIT OPTION

The E(xit option terminates the editing session without recording the changes made to the file currently in memory. Any changes made to the work file since the beginning of the editing session are NOT recorded. This option is useful when a file is to be read only.

RETURN OPTION

The R(eturn option returns to the Editor without recording any changes made during the editing session. The cursor returns to its location in the file at the time a Q was typed. This option is useful after a Q is inadvertently entered.

WRITE OPTION

The W(rite option provides the means to record the changes made during the editing session. The following prompt appears requesting the name of the file in which the changes should be recorded.

```
>Quit:
Name of output file (<cr> to return)-->
```

The changed file may be written to any file name. If the file already exists, the changed file replaces it. Typing a <ret> aborts the command.

Once the file name is entered, the following message and prompt appear:

```
Writing........
Your file is nnn bytes long.
Do you want to E(xit from or R(eturn to the Editor?
```

Typing an E exits the Editor and redisplays the outer level command prompt line. Typing an R returns the cursor to its previous location in the file. However, the changes made during the editing session were recorded on disk.

SAVE OPTION

The S(ave option is useful in the case where the Editor is used with a file other than the system work file. If the Editor is entered without a work file, the Editor prompts for the file to be edited. If a file name is entered, at Quit time, the S(ave option appears and asks if the file is to be saved as the name of the input file.

X(chng (Exchange Command)

The Exchange command (X(chng) allows existing characters to be exchanged on a one-for-one basis by new characters being entered. The Exchange command is initiated by typing an X from the Edit command line. The following prompt line then appears:

```
|>Exchange:Text{<bs>a char}[<esc> escapes; <etx> accepts]          |
```

As characters are entered, the cursor moves to the right over the text replacing the characters. If an <etx> has not been entered, backspacing restores the original characters on a one-for-one basis.

Typing an <esc> aborts the Exchange command without making the changes. Typing an <etx> (or editor accept key) accepts the changes as part of the file.

NOTE

> The exchange command does not allow typing past the end of the original text or the end of the line. Also, a <ret> may not be entered as a character to be exchanged. New text must be added through the I(nsrt command if the exchange exceeds the length of the original text.

After the Exchange command is initiated, the right arrow may be used to space over the existing text without changing it. Exchange is not affected by the current direction.

Z(ap (Zap Command)

The Zap command deletes all text between the start of the text last found, adjusted, replaced, or inserted and the current cursor position. Zap is designed to be used immediately after a Find, Replace, Adjust, or Insert.

| CAUTION |

> If any of the above commands are followed by a text change or any command that moves the cursor, the results of the Zap command are unpredictable.

The Zap command is initiated by typing a Z from the Edit command line.

If more than 80 characters are being zapped, the Editor asks for verification:

```
|>WARNING!You are about to zap more than 80 chars,do you wish to zap?(y/n)|
```

If the most recent text changing command was I(nsert, use of the Zap command deletes the insertion. If the most recent command was F(ind, use of the Zap command deletes the occurrence of the target found. If the most recent command was R(place, use of the Zap command deletes the substitute string from the text.

The text deleted is available for use with the C(py B(uffer command.

If the amount of text to be zapped exceeds the capacity of the copy buffer, the following message appears. (The maximum amount of text that can be zapped, and subsequently copied by the C(opy B(uffer command, varies depending on the size of the file being edited.) The = (equal sign) moving command jumps to the Z(ap "anchor" point.

```
---------------------------------------------------------------------------
|>WARNING!You are about to zap more than 80 chars,do you wish to zap?(y/n)|
---------------------------------------------------------------------------
```

If a Y is entered, the following message appears:

```
---------------------------------------------------------------------------
|There is no room to copy the deletion. Do you wish to delete anyway?(y/n)|
---------------------------------------------------------------------------
```

If a Y for yes is entered, the designated text is deleted and is not placed in the copy buffer. The designated text begins with the first character of the text last found, adjusted, replaced, or inserted; the designated text ends at the current cursor position.

Figures 3-5 and 3-6 present examples of the Zap command. The first example shows Zap used with the Find command to zap the target string. The second example shows the use of Zap after an insertion.

In the figures, user input is shaded. Comments are enclosed in braces ({}).

```
---------------------------------------------------------------------------
>Edit: A(djst C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(Chng, Z(ap ?

        {The following text contains the target to be found and zapped.}

        This paragraph illustrates the use of the Zap command to find a
        target and then remove it. The target to be zapped is the first
        occurrence of the word "command".

        F

        >Find [1]:L(it <target>=> / command/

        {The Find command searches through the file, placing the cursor
        at the end of the target.}

        Z {At this point, the target is zapped and the text is changed
           as below.}

        This paragraph illustrates the use of the Zap to find a target and
        then remove it. The target to be zapped is the first occurrence of
        the word "command".

              Figure 3-5. Use of Zap with the Find Command.
---------------------------------------------------------------------------
```

```
>Edit: A(djst C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap ?

                {The original text appears below.}

            1.  Turn on the power for the system terminal.
                Turn the round knob on the left side ofthe
                terminal clockwise until you hear a click.

                The intensity of the cursor and the characters
                are also changed byturning the knob.  The
                cursor is usually a rectangular box or an
                underline that moves over the screen to show
                you where you are currently keying on the screen.

            2.  Turn on the power for the system.  Press
                the white circle on the red switch that is
                located in the upper right corner of the
                system box.

    F
    >Find[1]: L(it <target> => /The intensity/

                {The Find command searches through the file,
                 placing the cursor at the end of the target.}

    Move the cursor to the end of the paragraph, past the word
    "screen.".  Then enter Z.

                {The following message appears:}

WARNING! You are about to zap more than 80 chars, do you wish to zap? (y/n)y

                {The zap is effected and the text is as below.}

            1.  Turn on the power for the system terminal.
                Turn the round knob on the left side ofthe
                terminal clockwise until you hear a click.


            2.  Turn on the power for the system.  Press
                the white circle on the red switch that is
                located in the upper right corner of the
                system box.
```

Figure 3-6. Example of the Zap Command.

Equal (=) Command

The Equal command is initiated by typing an equal sign (=) from the Edit main command line. Although not displayed on the Edit main command line, this command is displayed on the secondary Edit command line accessed by typing a ? from the Edit main command line.

This command moves the cursor to the beginning of the last portion of the text that was inserted, adjusted, found, or replaced. The Equal command is not direction-oriented; therefore, it is valid from any location in the file.

Whenever text is inserted, adjusted, found, or replaced, the beginning location is saved. However, if a copy or deletion is made between the beginning of the file and the absolute position, the beginning location of the last insertion, adjustment, find, or replacement is changed. Therefore, the Equal command location is no longer valid.

P(age (Page 3-Command)

The Page command displays the next page, whether forward or backward, where a page is the number of lines that are contained on the CRT screen (usually 23-24 lines). The cursor position remains the same except that its logical position is moved forward or backward by n lines.

At the end of the file, a complete screen is not displayed if the number of lines remaining is not a full page.

To move several pages at a time, the repeat-factor may be used.

The Page command is initiated by entering P from the Edit main command line. The Page command moves forward if a +P is entered or backward if a -P is entered. The last entry (+ or -P) remains in effect until changed by a subsequent explicit change of direction. That is, once a -P is entered, subsequent entries of P move backward in the file until a +P is entered. The forward direction is the default until changed by a -P entry.

Although not displayed on the Edit main command line, this command is displayed on the secondary Edit command line accessed by typing a ? from the Edit main command line.

M(rgn (Margin Command)

The Margin command adjusts a paragraph as closely as possible (without exceeding the margins) to the margins set in the Environment. The Margin command is initiated by typing an M from the Edit main command line with the cursor positioned somewhere within the paragraph to be adjusted. Although not displayed on the Edit main command line, this command appears on the Edit secondary command line, accessed by typing a ? from the Edit main command line.

A paragraph is defined as any text occurring between two blank lines. Additionally, a paragraph may delimited by the use of the Command Character as set in the Environment. In that case, the Command Character appearing as the first nonblank character on a line causes the Margin command to regard the line as a blank line. Therefore, the Margin command begins the paragraph on the line immediately after the line containing the Command Character and adjusts the text until the next blank line or line beginning with the Command Character is encountered.

The Margin command adjusts one paragraph at a time and is totally dependent on the right, left, and paragraph margins set in the Environment.

To margin a paragraph, the cursor is placed somewhere within the paragraph, and an M is typed. The Environment setting for A(uto indent must be false and the setting for F(illing must be true. The screen goes blank while the Editor is readjusting the paragraph. For a long paragraph, several seconds may elapse before the paragraph is redisplayed.

In breaking lines to avoid exceeding the right margin, the Margin command uses spaces or hyphens within words as breaking points. All other characters in sequences are considered to be words. Also, the Margin command may compress groups of spaces into single spaces.

Figure 3-7 gives two examples of the Margin command used with different margin settings. Figure 3-8 gives an example of the Margin command where the Command Character is used to delimit paragraphs.

```
>Edit: A(djst C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap ?
```

{The original text of the paragraph appears below.}

```
The Margin command is executed by typing an M when the cursor is in the
paragraph to be margined.  The Margin command adjusts only one paragraph
at a time and aligns the text to the specifications set in the Environment.
```

{To adjust the above paragraph to the margins — left = 10, right = 70, and the paragraph indentation = 10 — the specifications in the Environment must be set. Also, the A(uto indent option must be false, and the F(illing option must be true. Then the cursor is placed within the paragraph and the M typed. The resulting paragaph is shown below.}

```
          The Margin command is executed by typing an M when the
          cursor is in the paragraph to be margined.  The Margin
          command adjusts only one paragraph at a time and aligns the
          text to the specifications set in the Environment.
```

{In the following example, the paragraph is margined to the following specifications — left = 15, right = 75, and the paragraph indentation = 5.}

```
     The Margin command is executed by typing an M when the cursor is in
               the paragraph to be margined.  The Margin command adjusts
               only one paragraph at a time and aligns the text to the
               specifications set in the Environment.
```

Figure 3-7. Two Examples of the Margin Command.

```
>Edit: A(djst C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap ?

              {The text below is the original text before the Margin
               command is executed.}

    ^Baud Rate
    To set the baud rate for Port B using the H2 operating system with
    the SB1600, execute BAUD, a new program.
    ^Execution of BAUD causes a menu of baud rates for serial Port B to
    be displayed.  Select the number that corresponds to the appropriate
    baud rate.

    {The text below is the text after the Margin command is executed
     with the specifications -- left = 15, right = 65, and paragraph
     indention = 15.}

    ^Baud Rate
              To set the baud rate for Port B using the H2
              operating system with the SB1600, execute BAUD, a
              new program.
    ^Execution of BAUD causes a menu of baud rates for serial Port B to
    be displayed.  Select the number that corresponds to the appropriate
    baud rate.
```

Figure 3-8. Use of the Margin Command with Command Character.

S(et (Set Command)

The Set command offers two options: set markers or set environment. The Set command is initiated by entering an S from the Edit main command line. Although not displayed on the Edit main command line, this command appears on the Edit secondary command line, accessed by typing a ? from the Edit main command line.

The following prompt appears after the S is typed.

```
--------------------------------------------------------------------------
|>Set: E(nvironment M(arker <esc>                                         |
--------------------------------------------------------------------------
```

SET MARKER

Often in a long file the ability to jump to specified positions is a convenience. By setting markers in a file, the cursor can be moved quickly to those markers (Jump command). Markers can also delimit text in one file that is to be copied into another file (Copy command).

To place a marker in a file, the cursor is moved to that position and SM for S(et M(arker is entered. The following prompt appears:

```
--------------------------------------------------------------------------
| Set what marker?                                                        |
--------------------------------------------------------------------------
```

The name entered may be any length, however, at most, eight characters are recorded as the marker name. If the marker already existed, it is reset.

Only ten markers are permitted in a file at any one time. On typing SMzzz<ret>", if ten markers exist in the file, the following prompt and display appear, where the markers are named aaa through jjj. If the eleventh marker is one named already, the new marker is placed over the old one and no overflow prompt appears. For example, if "SMaaa<ret>" is entered at marker eleven, no overflow condition exists.

```
--------------------------------------------------------------------------
| Marker ovflw. Which one to replace?                                     |
| 0) aaa                                                                  |
| 1) bbb                                                                  |
| 2) ccc                                                                  |
| 3) ddd                                                                  |
| 4) eee                                                                  |
| 5) fff                                                                  |
| 6) ggg                                                                  |
| 7) hhh                                                                  |
| 8) iii                                                                  |
| 9) jjj                                                                  |
--------------------------------------------------------------------------
```

When a number between Ø and 9 is entered, that space is available for setting the desired marker. Thus, one existing marker must be removed in order to add another one.

A marker may be removed by deleting the text that contains the marker.

SET ENVIRONMENT

The Set Environment command allows the editing environment to be controlled by the user. This command offers options pertaining to text formatting; also the Set Environment command displays other information regarding the file being edited.

The Set Environment command is initiated by typing an S and then an E (or an SE) from the Edit command line. A display similar to the following appears:

```
>Environment: {options}<etx> or <sp> to leave
    A(uto indent  True
    F(illing      False
    L(eft margin  Ø
    R(ight margin 79
    P(ara margin  5
    C(ommand ch   ^
    T(oken def    True

    nnn bytes used, nnnnn available

    Patterns:
      <target>= '    ', <subst>-

    Markers:
      ORANGE    PEACH
```

The options shown in the upper part of the display are changed by entering the first letter of the option name and the new value.

These options are explained in the following paragraphs. The number of bytes used in the file is shown as is the number of bytes remaining for use. The creation date of the file and the date that the file was last written are also shown.

The "Patterns" information does not appear unless a Find or Replace has been completed during the editing session. The "Markers" information is not displayed unless markers exist in the file.

A(uto indent - When true, the auto indent option causes each new line inserted to be aligned (or indented) with the first nonblank character of the previous line. When false, this option causes the lines to be aligned with the left edge of the screen or begun at the Left margin set in the environment. This option affects the results of the Insert and Margin commands.

In order for an Insert to automatically remargin at the part of the paragraph following the insertion, A(uto indent must be set to false and F(illing must be set to true. Likewise, the Margin command requires those settings in order to adjust an entire paragraph.

To set the A(uto indent option, either AT or AF is typed. If any character (or numbers other than T or F is entered, a beep sounds and "T or F" appears to the right of the option. Also, the screen is frozen until an appropriate choice is made.

F(illing - When true, the lines of text are automatically filled with words up to the right margin set in the Environment. Lines are broken between whole words (at spaces) or at hyphens only.

When false, the margins set in the Environment are ignored; the text is spaced as entered.

This option affects the results of the Insert and Margin commands. The Insert command does not cause remargining of the paragraph text following an insertion if F(illing is false. Likewise, the Margin command cannot adjust a paragraph if F(illing is false.

To set the F(illing option, either FT or FF is typed. If any character (or number) other than T or F is entered, a beep sounds and "T or F" appears to the right of the option. Also, the screen is frozen until an appropriate choice is made.

L(eft, R(ight, and P(aragraph Margins - These margins are set by entering an L, an R, or a P plus an unsigned integer less than or equal to 84 (the maximum screen width supported by the Editor) and a <space> or <ret>. These settings are used by the Insert and Margin commands in adjusting a paragraph. These settings also affect the center and justify options in the Adjust command.

C(ommand character - This command also affects the Insert and Margin Commands. (See the discussion of Command Character in the Margin Command section.) When the Command Character is the first nonblank character on a line, that line is not subject to remargining by the Insert or Margin commands.

The default Command character is the caret (^). This character appears as an up arrow (^) in the Environment display but prints as a caret.

T(oken default - The setting of this option affects the search type used of the Find and Replace commands. (See those sections for additional explanation.)

If the T(oken default is set to true by entering TT, the default search for the Find and Replace commands is a token search. If the T(oken default is set to false by entering TF, the default search for the Find and Replace commands is a literal search.

If any character (or number) other than T or F is entered, a beep sounds and "T or F" appears to the right of the option. Also, the screen is frozen until an appropriate choice is made.

V(erify (Verify Command)

The Verify command verifies the contents of the work file and the Editor status by redisplaying the screen. The Verify command is initiated by typing V from one of the Edit command lines. Although not displayed in the Edit main command prompt line, this command appears on the secondary Edit command line, accessed by typing a ? from the Edit main command line.

This command redisplays the text window and attempts to adjust the window so that the cursor is at the center of the screen.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

3.2 ADVANCED EDITOR

The Advanced Editor is an enhanced version of the Screen-Oriented Editor that offers additional capabilities and some efficiencies in text manipulation. The Advanced Editor is included in the H3 III.Ø Operating System as an alternative to the regular Screen-Oriented Editor, which is named SYSTEM.EDITOR on the operating system.

| NOTE |

> If the Advanced Editor is to be used as the system editor, the Screen-Oriented Editor can be renamed, and then the Advanced Editor can be named SYSTEM.EDITOR. In that case, the Advanced Editor is called when an E (for E(dit) is entered from the system command line.

The Advanced Editor is directed to experienced users of the MicroEngine. This editor is intended for use in a program environment where large programs with multiple include files are used. The Advanced Editor allows editing of different files without the need to go to the Filer to change or save the work file. In addition, the Advanced Editor supports a macro capability so that powerful strings of editing commands can be invoked by a one character command.

Because the Advanced Editor is self-documenting (interactive documentation), Section 3.2.1 presents the sections of the interactive documentation as they appear on the terminal display. Section 3.2.2 summarizes the extentions and differences between the Screen-Oriented and Advanced Editor commands.

3.2.1 Interactive Documentation

The following information appears on the display when the interactive documentation is accessed. In the Advanced Editor, the documentation is accessed by typing a '?' from the edit command line. The various sections and subsections of the interactive documentation are listed continuously in this book, separated by dotted lines.

When accessed interactively, the various sections are chosen by the user depending on the information required.

Advanced Editor Commands

Moving	J(ump [B(egin, E(nd, M(arker, A(djourn], <left>, <right>, <up>, <down>, '=', F(ind, W(ord, P(age, <space>, <ret>, <tab>, [directional] '>', '.', '+' or '<', ',', '-' [set direction]
Formatting	A(djust, M(argin
Text changing	D(elete, I(nsert, R(eplace, X(change, Z(ap, C(opy [B(uffer, F(ile]
Control	C(opy [C(ontrols], S(et [E(nvironment, M(arker, A(djourn, *(macro], V(erify [redisplay screen], Q(uit [buffer action, next editing option], '?' [interactive documentation]

Interactive Documentation

Interactive documentation in the Advanced Editor is requested by '?' at most editor prompts that select command options. '?' at the outer level displays all commands and allows selection of the Advanced Editor introduction, this section, or specific documentation on any outer level command. '?' at any other editor prompt displays documentation for that command.

Documentation is organized into sections, some of which have subsections presenting further information. After any section other than the outer level section is displayed, the next option prompt offers <esc> or selection of the "parent" section or any subsection of the current one. The prompt after the outer level section offers the options mentioned above.

<esc> returns to the editor prompt where the '?' was initially entered. Parent and subsections are selected by moving the cursor to the section title and typing <etx>. The cursor is intially placed on the Parent title (the section of which the current one is a subsection).

If a section requires more than one screen page, the following prompt occurs before scrolling to the next page. If reading this section for the first time, please type <etx>.

<esc> or <etx> to continue

After scrolling has occurred, typing <ret> to any subsequent prompt within the section causes it to be redisplayed from the start. This special option does not appear in prompts.

Advanced Editor Introduction

The Advanced Editor is an extension of the Screen-Oriented Editor. It increases the range of text which may be conveniently edited in one session, while providing more flexible editing control of the current file. A summary of areas of added capability includes

- improved file control and protection,
- refinement of some basic editor commands and addition of others,
- user-defined editing commands by use of a flexible macro facility,
- multiple file editing with Pascal work file control commands, and
- interactive documentation.

To read about interactive documentation, type <etx> to return to the complete command display and then type '?'. Or type any Advanced Editor command listed there for specific command documentation. Select the File Control section here for discussion on expanded editing controls and multiple file editing, or the Introduction to Macros to find out how to define your own editing commands.

File Control and Protection

UCSD Pascal text files include a header record that stores editing control information for the file. The Advanced Editor adds new control capabilities, including user-defined editing macros, an adjourn (automatic return) point in the file and flexible tab stops. It allows copying controls between files and initializes new files to a standard set of controls provided by the user.

Multiple files may be edited during one session, and for each file the Advanced Editor records whether or not the text and/or the editing controls have been changed. This status is shown with the current file name in the S(et E(nvironment display and by the Q(uit display.

The Q(uit command presents edit buffer action and next edit options according to this information. It also guards against inadvertent loss of editing changes. The Pascal work file may be changed by the buffer action, or files other than the work file may be edited without affecting the Pascal work file status.

Editing Controls

The edit-controls header in each text file contains the controls defined below. The Advanced Editor has added control capabilities; this change is reflected by a different header version number from the two values used by the Screen-Oriented editor and the L2 editor.

Controls can be copied to and from text files and "edit-controls only" data files. These data files contain only the controls header and may be used to store the different controls appropriate for documents, program source, etc. One such file, '*ADV_ED.CONTROLS', is assumed to be a user's standard set of controls. The C(opy C(ontrols command accepts the name '*' as shorthand for this file.

The Advanced Editor automatically initializes any new file or a text file with an old control version from the standard controls '*ADV_ED.CONTROLS'. If it is not present, the file contains no markers or macros, and the other control values are set to default values defined below. Controls other than Macros, Markers, and Adjourn are changed by S(et E(nvironment.

Macros - User-editing commands defined by S(et *(macros. See 'Introduction to Macros' for further discussion.

Markers - Up to ten, named, cursor locations in the file. Names are eight characters in significance. Markers are set by S(et M(arker and jumped to by J(ump M(arker. If text containing a marker location is deleted, the marker is removed. The C(opy F(ile command allows use of markers to delimit the text to be copied. Markers are related to the text in the file and are not copied by C(opy C(ontrols.

Adjourn - An explicit location to which the cursor is positioned on entry. It is set by S(et A(djourn and jumped to by J(ump A(djourn. Its default location is the start of the file. If text containing the Adjourn location is deleted, it is reset to the start of the file. Like markers, this value is not copied by C(opy C(ontrols.

Tabs - User-selected tab stops at any column. Tabs are jumped to by the directional moving command <tab>. Default tab stops are 0,8,16,...

Auto-indent - A Boolean affecting Insert. If true, new line indentation is aligned with the previous line; otherwise, the Left Margin is used. Default is true.

Filling - A Boolean affecting Insert. If true, <ret> is automatically added to keep text within the Right Margin. The Margin command requires Filling true. Default is false.

Token def - A Boolean affecting Find and Replace. If true, the default pattern match mode is token; otherwise, literal. See 'Find command' section for further definition. Default is true.

Left, Paragraph, Right Margin - integers affecting Insert, Adjust, and Margin. Values are column numbers in the range Ø..79 for an 8Ø-column terminal. The Paragraph margin is the indentation of the first line. Defaults are Ø,5,79.

Quit Command

The Quit command normally displays the current file name, whether or not controls and/or text have been changed and shows two prompts: buffer action and next options. Buffer action refers to options available regarding the current edit buffer, and next options offer the user a choice of subsequent actions, including editing another file. No action is taken by the Advanced Editor until a next option other than backspace is entered.

The buffer action prompt is presented only when the text in the file has been changed. If no changes are made, Quit displays 'no changes to' file name instead of the buffer action prompt and offers the next option prompt. If only control changes are made, Quit similarly displays 'Control changes only' with the file name and will update the changes in the source file by default.

The user may backspace from the next option back to the buffer action. For example, if default update of control changes is not desired, the changes may be discarded by backspacing from the next option to the buffer action and then entering D(iscard.

To protect against inadvertant loss of editing changes, the Advanced Editor always requires confirmation before text changes are discarded.

If a code file with the same name as the text file exists, it is removed when text changes are written to the source file. The code file is not removed if only control changes are written to the source file.

```
<file-name>           Buffer Action ?
<  status >           ---------------
                      '$', <esc>, <ret>, '?'
                      U(pdate
                      S(ave
                      W(rite to a file
                      D(iscard
```

The action to be taken on the edit buffer is specified by this prompt. The source file name (or 'new file') and its status are shown to the left of the prompt. Status indicates whether or not text and/or controls changed.

```
<esc>     - Leaves the editor.
<ret>     - Returns to editing.
D(iscard - Discards edit buffer and offers the next option.
```

If text changes are made, <esc> and D(iscard confirm:
    'Discard changes to <file-name> ? '
N(o requires respecifying the buffer action.

```
'$'      - Writes the buffer to the source file without altering the Pascal
           work file. This option is not available for a new file.
U(pdate - Writes the buffer to '*SYSTEM.WRK.TEXT' and updates Pascal work
           file status to reflect this file as the current version, with the
           source file as the base file.
S(ave    - Writes the buffer to the source file and updates Pascal work file
           status to reflect the source file as the base file. Existing
           updated versions of the Pascal work file are removed. For a new
           file, the base file name is entered after the prompt
               'Save as what file ? '
W(rite   - Writes the buffer to the file name specified after the prompt
               'Output file ? '
           If the file name is not a disk file (for example, 'PRINTER:'),
           the file header containing the edit controls is not written.
```

If an updated work file ('*SYSTEM.WRK' - text and/or code) exists whose base file is different from the current source file, U(pdate and S(ave must remove it. In this case, both confirm:
    'Discard current work file (source = <file-name>) ? '
N(o requires respecifying the buffer action.

==> Next Options

<esc>, <ret>, <left>, '?'
<file-name>, '*'

Next Option ?

The action to be taken next is specified by this prompt. No buffer action is taken until a response other than <left>, or backspace, is given. After the next option is specified, the buffer action (if any) is performed before the requested next action.

<esc> - Leaves the editor.
<ret> - Returns to editing.
<left> - Goes back to the buffer action prompt for respecification.

<file-name> - Causes the text file name entered to be edited next.
'*' - Causes the most recent version of the Pascal work file to be edited next. This file is '*SYSTEM.WRK.TEXT' if present or the base file otherwise. This option is not offered if no Pascal work file exists.

If a <file-name> is entered but the file is not present, Quit prompts
'Create <file-name> ? '
If the answer is Y(es, a new file with the given <file-name> may be created next. N(o requires respecifying the next option.

Editor Input Conventions

Outer level editor commands are selected by typing single terminal keys or a special Prefix and a single key. Each key except the Prefix may be associated with a primary and an alternate command, and either command may be a basic editor command or a user-defined macro command. Typing a key invokes the primary command and typing Prefix key invokes the alternate command. See 'Introduction to Macros' and its subsections for further discussion.

Upper and lower case letters are treated as the same key at the outer level and in all cases of single char command options.

Basic editor commands commonly prompt the user to enter a name or select a command option. Command options are given by typing one of the displayed single chars. The entered char is not interpreted as a macro, although a previously invoked macro may include the single char response in its macro string.

File or marker names are entered by typing the name and <ret> to terminate the input. <left>, or backspace, is used to back up over the previous char. <del> backs up over all the previous input. The particular command is aborted if the name is empty when <ret> is typed, or if <esc> is typed at any time before <ret>. The standard suffix '.TEXT' does not need to be typed for text file names; it is appended to the name as needed.

User confirmation of a particular action and other yes or no questions are offered by various editor commands. These are answered by Y(es, N(o, or <esc>. Normally <esc> is the same as N(o; differences to that rule are explicitly noted in the documentation.

Introduction to Macros

Macros are user-defined editing commands. Each macro is a key which maps to a string of up to 25 characters. The string defines the editing action that takes place when the macro is used. Macro command keys are recognized at the outer level of the editor and in places where moving commands are valid (within D(elete mode for example).

The macro string may include basic editor commands, responses to prompts that result from those commands, or macro keys (including recursion on the macro itself). Macros may also be defined to include interactive input. A special char (shown as <user>) may be put into the macro string. When this char is encountered, input is taken interactively until <etx> is typed.

Any key on the terminal may be defined to be a macro command. To increase naming flexibility, the Advanced Editor can recognize two commands for any key: the primary and alternate commands. Either of the commands associated with a key may be a macro or a basic editor command. For example, a macro named 'D' may be defined as the alternate 'D' command, where D(elete remains the primary command, or vice versa.

A special "alternate command" key called the Prefix is used preceding any command key to indicate the alternate command for that key. The Prefix key is specified by the user and may be changed at any time.

Macros are stored in the edit controls header of each text file, so that any file may contain its own set of macros. Easy copying of macros among files is provided by the C(opy C(ontrols command, as well as the capability for each new file to be initialized from a standard set of macros (discussed in the 'Edit Controls' section).

Macros are displayed, defined, and removed by the S(et *(macros command. This command could not be named S(et M(acros because that would have conflicted with S(et M(arker.

Macro Command Invocation

Macro commands are recognized at the outer level and where moving commands are valid. The macro string is "expanded" into a special input buffer, and input is taken from the macro expansion until it is exhausted. Nested macro keys are similarly expanded when encountered, including recursion on the original macro. Dynamic macro expansion may be up to 255 characters.

When the <user> input char is encountered in the macro expansion, the active macro is suspended, and input is taken interactively from the user. This mode is terminated by typing <etx>, causing resumption of the macro. The <etx> is not read by any editor command; it only switches input mode.

The Prefix key is used within a macro to specify an alternate command just as it is interactively. The following example shows definition of a macro as the primary command such that a basic editor command becomes the alternate command. The example also illustrates a <user> input parameter and nesting of macros. The default Prefix value '@' is used in the example.

Macro	Definition	Explanation
@B	JB	J(ump B(egin
F	@B@F/<user>/	@B invokes the jump begin macro, @F is the alternate F command (the standard Find) and <user> allows the pattern to be typed interactively. After the terminating <etx> is typed, the closing delimiter / is read from the macro to initiate the find.

The outer level input 'Fwhat_ever<etx>' jumps to the beginning of the file and finds the pattern 'what_ever'.

Macros: D(efine, R(emove, C(ontrol-chars, '?', Q(uit

The macro environment accessed by S(et *(macros displays the above prompt, the current Prefix and <user> characters, how many macros are defined and available, and all the currently defined macros. Prefix and <user> characters may be changed by the C(ontrol-chars command; a total of 20 macros may be defined for any file. Macros are always displayed with the Prefix if they are alternate commands:

<key> = <macro-string>   or   @<key> = <macro-string>

The <macro-string> shows printable characters directly and nonprintable characters as <name> or CTRL_<key>. Common keyboard characters are shown in the first form (<ret>, <left>, <etx>). Otherwise, they are shown in the CTRL_ form, where <key> is the appropriate ASCII char typed with CTRL.

R(emove prompts 'Remove what macro (CTRL_E to escape) '; the macro to be removed is entered (with the Prefix if an alternate command). See 'Macro Control Characters' section for explanation of the macro escape key.

Controls: P(refix, A(ccept, E(scape, B(ack-up, U(ser-input, '?', Q(uit

@ = Prefix char
CTRL_A = Definition accept char
CTRL_E = Definition escape char
CTRL_B = Definition back-up char
CTRL_U = User interactive-input char

The C(ontrol-chars command displays the above prompt and control-char status. The values shown here are the defaults. Each may be changed by entering the desired control-char name, for example A(ccept, and then the new value.

The Prefix and <user> chars are discussed in 'Introduction to Macros'. Both can be included in defined macros, and when their values are changed by this command, all current macros are automatically updated to the new value.

The other three control characters are used in defining macros. The escape char may also be used to abort the R(emove macro command or any of the above control-char change commands.

Define what macro (CTRL_E to escape)

The macro <key> is entered, preceded by the Prefix if it is to be an alternate command. The definition escape char may be used to abort the new definition. If the <key> is not an existing basic editor command or macro, Define shows the control characters used for definition and the new macro:

CTRL_A accept, CTRL_E escape, CTRL_B back-up, CTRL_U user-interactive-input
<key> =

The macro string is then entered. The accept char is used to terminate the definition; the escape char to abort it; and the back-up char backspaces over the last char entered (default values are shown above).

If the new macro <key> is a basic editor command or a macro, Define prompts as follows before the macro string is entered:

```
    Override <editor command> ?        (for example 'Override F(ind ?')
        or
    Replace macro ?
    <key> = <macro-string>
```

These prompts may be answered Y(es, N(o, or <esc> (the normal editor <esc>). Y(es causes the <editor command> to become the alternate <key> command or implicitly removes the previous macro. <esc> aborts the definition. If the new macro was not initially prefixed, N(o prompts

```
    Prefix <key> ?
```

If Y(es is entered, <key> is defined as an alternate command; otherwise, the new definition is aborted.

Set: E(nvironment, M(arker, A(djourn, *(macro, '?', <esc>

The Set command offers access to edit controls according to the above prompt. See subsections for Set Environment and Set Macro command discussions.

M(arker - Prompts for a marker name:
'Set what marker? '
The marker entered is set to the current cursor location. The marker name may contain any char and is significant through eight chars. A file may have up to ten markers. If no markers are available and the entered marker does not already exist, the ten existing markers are displayed with the prompt
'Marker overflow: enter name of marker to replace or <esc> '
An immediate <ret> or <esc> before <ret> for either prompt aborts the Set Marker command.

A(djourn - Sets the adjourn location to the current cursor position. This causes the cursor to be automatically set to its current position when the source file is next edited.

Environ: <option letters>, S(et-tabs, '?', Q(uit

The edit control environment accessed by S(et E(nvironment displays the above prompt, the Advanced Editor version, edit control values that may be changed by environment options, the source file name and edit buffer status, and other control information. See 'Edit Controls' section for definition of control values.

<option letters> refer to the first letters of the edit controls shown at the top of the screen. Their values are changed by entering the appropriate char and the new value. Auto indent, Filling, and Token def are Booleans; new values are entered as 'T' or 'F'. Command ch is set to any char value. Left, Right, and Paragraph Margins are integers in the range [Ø..84].

The source file name is displayed; if text and/or edit controls are changed, the change status is shown to the right of the file name. Also displayed is the number of edit buffer bytes currently used and available, and the date the source file was created and last changed.

If markers exist in the file, their names are displayed in the order in which they occur (from the start to the end of the file). Find/Replace patterns are displayed if they have been defined. <target> is the last Find pattern or Replace source pattern, and <sub> is the last Replace substitute pattern. The number of <target> replacements by <sub> is shown after Replaces.

Tabs are displayed as a full line of '-' and 'T' chars, where 'T' indicates a column with a tab stop. The S(et-tabs option positions the cursor on column Ø of the tab line and prompts as follows:

Tab stops: Q(uit, <left,right>, S(et, R(eset, Z(ero, C(ol # Ø

T-------T-------T-------T-------T-------T-------T-------T-------T-------T------

The cursor column position is changed by entering an optional repeat-factor with <left> or <right>, or by entering C(ol and the new column value. S(et and R(eset define and remove a tab stop at the current column; 'T' and '-' are also recognized for S(et and R(eset, respectively. Z(ero resets all tab stops from the current column to the end of the line.

Copy: B(uffer, F(ile, C(ontrols, '?', <esc>

Text and editing controls are copied according to this prompt. The F(ile and C(ontrols options provide copying from the current edit buffer to external files or from external files into the edit buffer. See the respective sub-sections for further discussion.

The B(uffer option specifies copying text from the copy buffer to the current cursor location. The copy buffer contains the text last inserted, deleted or zapped. It contains the deleted text if delete is terminated by <etx> or the text that would have been deleted when delete is terminated by <esc>. The two delete cases are effective ways to move and copy text respectively. The copy buffer is not changed if insert is terminated by <esc>.

If the copy buffer contains entire lines, all lines are copied with their original indentation before the line in which the cursor is located. If the copy buffer contains partial lines, it is copied to the exact location of the cursor and the line into which it is copied retains its current indentation.

Copy controls F(rom T(o another file, '?', <esc>

Editing controls are copied according to this prompt. The first option specifies the direction of the copy: F(rom another file into the current edit buffer or T(o another file from the edit buffer. The copy controls command prompts according to the specified direction

TO what file ('*') ? or FROM what file ('*') ?

where '*' stands for the user's standard set of controls, '*ADV_ED.CONTROLS'. The file name entered may refer to a text file or an "edit-controls only" data file, such as '*ADV_ED.CONTROLS'. See 'Editing Controls' section for further information on edit controls files.

When the command is C(opy C(ontrols T(o and the file does not exist, an edit controls data file is created with the entered file name. This case allows storing a set of editing controls for subsequent copying by other files. It is the means of initially creating the standard controls file.

Markers and the Adjourn location are directly related to the text in the file and are not copied by the copy controls command.

Copy text F(rom T(o another file, '?', <esc>

Text is copied according to this prompt. The first option specifies the direction of the copy: F(rom another file into the current edit buffer or T(o another file from the edit buffer. The copy file command prompts according to the specified direction

TO what file (from <marker-spec>) ? or FROM what file (<marker-spec>) ?

The copy TO option creates a file with the entered file name containing text from the current edit buffer. The optional <marker-spec> may be used to delimit the text that is written. The copy FROM option copies text from the entered file to the current cursor location; in this case, the optional <marker-spec> refers to markers in the external file.

The <marker-spec> includes marker names enclosed by '[' and ']':

```
<marker-spec> = [ M ,   ] - from marker M to the end of the file
                [  , M  ] - from the start of the file to marker M
                [ M1,M2 ] - between markers M1 and M2 (order doesn't matter)
```

C(opy F(ile T(o allows a fourth <marker-spec> form:

```
                [   M   ] - between the current cursor location and marker M
```

In all cases, the copy file command copies entire lines. Text is copied from the start of the line containing the initial marker (cursor) to the end of the line containing the final marker (cursor).

Non-Directional Moving Commands

These moving commands are not affected by the current direction. The arrow keys may be preceded by a repeat-factor to specify the number of columns or lines to move. The Jump command is discussed in a separate section.

```
<left>  - moves the cursor x columns left   | x = 1
<right> - moves the cursor x columns right  | or
<up>    - moves the cursor x lines up       | x = repeat-factor
<down>  - moves the cursor x lines down     |
```

<left> and <right> keep the cursor within lines; <left> moves from the start of a line to the end of the preceding line and <right> moves from the end of a line to the start of the next line. <up> and <down> maintain the current column in the line to which the cursor is moved.

'=' - moves the cursor to the start of the text last Found, Replaced, Inserted, or Adjusted; the Zap command deletes text from '=' to the current cursor position.

Jump: B(egin, E(nd, M(arker, A(djourn, '?', <esc>

The Jump command repositions the cursor according to the above prompt:

B(egin - Jumps to the start of the first line in the file.

E(nd - Jumps to the end of the last line in the file.

M(arker - Jumps to the location of the marker entered in response to 'Jump to what marker? '
An immediate <ret> or an <esc> before <ret> aborts the jump.

A(djourn - Jumps to the adjourn location in the file. This is the initial cursor location when the file is next edited.

Directional Moving Commands

These commands move the cursor in the current direction by units depending on the particular move -- chars, lines, etc. Each directional moving command may be preceded by a repeat-factor that specifies the number of units to move. F(ind is also a directional command; it is discussed in a separate section.

<space> - moves chars (columns)
<ret> - moves lines and positions the cursor on the start of the line
<tab> - moves to tab stops
W(ord - moves words and positions the cursor on the start of the word -- words are sequences of chars separated by <space>s and <ret>s
P(age - moves screen display pages and redisplays with the cursor on the same relative screen line on which it was initially located

The default direction is forward; the following commands change direction

'<', ',' and '-' set direction backward, or
'>', '.' and '+' set direction forward.

>Find[r]: '?' L(it <target> => or >Find[r]: '?' T(ok <target> =>

The Find command is a directional moving command which positions the cursor to the end of a specified <target> pattern. Find offers two pattern matching modes: Token and Literal. The Token def control value determines the default mode; the other mode is selected by entering L(it or T(ok preceding the <target>. The <target> pattern used by the last Find can be specified by entering 'S' (for Same) instead of the full <target>.

Literal mode causes any occurrence of the <target> pattern to be found exactly as entered, including multiline patterns. Token mode causes isolated occurrences of tokens to be found. A token is any punctuation char or a name (a sequence of alpha, numeric, and '_' chars.) Names are delimited by <ret>, <space>, or punctuation tokens. Multitoken patterns can be specified.

A repeat-factor is used with Find to specify how many occurrences of the <target> pattern to find before stopping. The default value of 1 causes the next <target> pattern in the current direction to be found. The Find prompt shows the repeat-factor, indicated by '[r]' above.

<target> Specification

The <target> pattern is enclosed within a set of identical delimiters. The first char entered defines the delimiter, which can be any char other than a token name char. The <target> pattern is the chars entered between (but not including) the delimiters. L(it or T(ok is entered before the opening delimiter to switch the default pattern mode.

<left> backspaces over the preceding <target> char, and <del> backs up to the start of the <target>.

'S' is entered instead of the <target> to indicate the same pattern. All occurrences of a given <target> can be found by entering the full <target> initially and 'FS' to find subsequent occurrences. L(it or T(ok must be entered as needed with each Find command; for example, if Token def is true and the initial Find is 'FL<target>', subsequent literal Finds must be entered as 'FLS'.

Aborting Find by <esc>

<esc> can be entered at any time before the closing delimiter to abort the Find command; in this case, the previous <target> is not changed. <esc> can also be entered to abort the Find during the <target> search.

Multiple-File Finds

The current <target> pattern is preserved throughout an editing session. This feature allows entering a pattern once, using the Q(uit <file-name> option, and using 'FS' to search for it in other files.

Literal / Token Mode Examples

The literal <target> /is/ is found in any of the following text sequences: 'isolated', 'find is safe', or 'distance'.
A token mode Find of /is/ finds only the middle occurrence. A token Find of /x is/ also finds 'x is ' or 'x
 is not'.
A literal find of /x is/ finds neither occurrence.

Adjust[xx]: L(just, R(just, C(enter, <arrows> {<etx> to leave}

The Adjust command allows lines of text to be shifted right or left according to the above prompt. Options other than <up> and <down> refer to the line in which the cursor is located. To adjust a sequence of lines, one line is adjusted; then <up> or <down> is used to adjust the line above or below by the same amount. The cursor column is shown in the prompt throughout the Adjust command (indicated by [xx] above).

Adjust mode is terminated by entering <etx>; <esc> can only be entered to abort the Adjust command before any line adjustment is specified. Specific adjust options are defined below; a repeat-factor can be used with the <arrow> keys (x is used below to mean 1 or the repeat-factor).

L(just - aligns the current line to the Left Margin
R(just - aligns the current line to the Right Margin
C(enter - centers the current line between the Left and Right Margins
<left> - shifts the current line x spaces left
<right> - shifts the current line x spaces right
<up>   - adjusts x lines above the current line by the current adjust amount
<down> - adjusts x lines below the current line by the current adjust amount

---

Margin Command

The Margin command is used to adjust the paragraph in which the cursor is located as closely as possible to the Paragraph, Left, and Right margins. A paragraph is defined as any text occurring between two blank lines. A paragraph may also be delimited by use of the Command Char appearing as the first nonblank char on a line. In that case, the line is regarded as a blank line.

The first line of the paragraph is adjusted to the Paragraph margin; other lines are adjusted to the Left margin. In breaking lines to avoid exceeding the Right margin, the Margin command regards <space>, <ret>, and hyphen ('-') as word delimiters. Also, the Margin command may compress multiple <space>s into a single <space>.

Margining is normally used when Auto-indent is false and Filling is true; if either value is not as above, Margin prompts
    'Do you wish to margin this paragraph ? '
If the response is Y(es, the current paragraph is margined; otherwise the Margin command is aborted. (Auto-indent and Filling also affect Insert).

All control values that affect Margin are accessed by S(et E(nvironment.

Insert: TEXT, <left>, <tab>, <del> {<etx> to accept, <esc> to abort}

The Insert command allows general insertion of text into the edit buffer according to the above prompt. The new text is inserted as typed at the location of the cursor when Insert is entered. Insert mode is terminated by <etx> to accept or <esc> to abort the insertion.

If Auto-indent is true, new lines are aligned with the preceding line; otherwise new line indentation is the Left Margin. If Filling is true, <ret> is automatically added to keep lines within the Right Margin. The following special chars are interpreted as Insert control commands; other nonprintable chars are inserted as typed and displayed as '?'.

<left> - backspaces over the preceding inserted char
<del> - backs up to the end of the preceding inserted line
<dcl> - backs up to the start of the current inserted line
(<dcl> is ASCII code 17 — normally CTRL_Q)
<tab> - inserts blanks up to the next tab stop on the line

The inserted text is available to be copied by C(opy B(uffer if accepted; if Insert is aborted by <esc>, the current copy buffer is not changed.

Delete, Zap commands

These two commands delete text from the edit buffer. The deleted text is available for subsequent copying by the C(opy B(uffer command.

Delete - Enters Delete mode; the current cursor position is recorded as the anchor location and the following prompt is displayed

>Delete: < > <Moving commands> {<etx> to delete, <esc> to abort}

All moving commands may be used in Delete mode, including direction change. Text is deleted as the cursor is moved away from the anchor and restored as the cursor is moved toward the anchor. Delete mode is terminated by <etx> to accept or <esc> to abort the deletion. If <esc> is entered, the text which would have been deleted by <etx> is available for copying.

Zap - Deletes text between the cursor and the '=' location -- the start of the text last Found, Replaced, Inserted, or Adjusted. If more than 80 chars are being zapped, Zap requires confirmation before deleting the text.

>Repl[r]: '?' L(it V(fy <target><sub> =>
or
>Repl[r]: '?' T(ok V(fy <target><sub> =>

The Replace command finds a <target> pattern and replaces it with a specified substitute. Replace extends the Find command by offering pattern replacement capability. See the Find command section for discussion of all aspects of <target> pattern search, including Token and Literal pattern matching modes. Token def, accessed by S(et E(nvironment, determines the default mode.

A repeat-factor is used with Replace to specify how many <target> patterns to replace by substitute patterns. '/' can be used with Replace to specify all occurrences. The prompt shows the repeat-factor, indicated by '[r]' above.

The Replace command searches for the <target> pattern in the current direction, finds the specified number of occurrences, and replaces each occurrence with the substitute pattern. User verification of each replacement is optionally requested by entering V(fy preceding the <target> pattern. When verification is selected, Replace offers the following prompt for each <target> occurrence

>Replace[r]: <esc> aborts, 'R' replaces, ' ' doesn't

<esc> is used to abort the Replace command. 'R' causes replacement of the current <target> occurrence, and ' ' indicates not replacing it. The Replace command continues after both 'R' and ' '.

The <target> and substitute patterns are entered according to the rules defined in the Find command section. The substitute pattern is indicated by <sub> in the prompt. Each pattern is enclosed within a pair of identical delimiters. The delimiters used for the <sub> pattern can be different from those used for the <target> pattern.

The options L(it or T(ok, and V(fy are entered before the <target> pattern or between the closing delimiter for <target> and the opening delimiter for <sub>. 'S' can be used for either pattern to indicate the same pattern as the last Replace. The S(et E(nvironment command displays the current form of both patterns and shows the number of <target> patterns actually replaced after a Replace command is executed.

Xchange: TEXT, <left>, <right> {<etx> to accept, <esc> to abort}

The Xchange command allows existing chars in the current line to be exchanged on a one-for-one basis by new chars being entered. New chars are exchanged at the current cursor location and the cursor is moved right for each char entered.

The <left> arrow, or backspace, moves the cursor and restores the original chars. The <right> arrow moves the cursor over existing chars without exchange. Xchange mode is terminated by <etx> to accept or <esc> to abort the exchange.

If the cursor reaches the end of the line, the Xchange is implicitly accepted and the Insert command is automatically invoked. This allows easy extension of the length of the line or addition of new lines following it.

Verify command

The Verify command redisplays the screen without moving the cursor. If no repeat-factor is entered, the line containing the cursor is centered in the redisplayed screen.

A repeat-factor is used with Verify to specify the redisplayed screen line number for the line containing the cursor. This allows explicit control over the amount of text that is shown before and after the cursor.

A repeat-factor in the range [2..height] is a valid line number, where height is 23 for a normal 24 line terminal. Prompts are displayed on line Ø; the default repeat-factor, 1, and other invalid values cause centering.

3.2.2 Command Differences

The following summaries briefly describe the differences between specific Advanced Editor commands and the corresponding Screen-Oriented Editor commands. In most cases, the Advanced Editor command extends the capability of the Screen-Oriented Editor command. The commands are grouped as follows: moving commands, formatting commands, text-changing commands, and control commands.

Refer to the appropriate discussions in Section 3.2.1 for details of the Advanced Editor commands and to Section 3.1 for discussions of the Screen-Oriented Editor commands.

Moving Commands

Several moving commands are extended (or different) in the Advanced Editor.

JUMP COMMAND

In the Advanced Editor, the Jump command contains an additional option - J(ump A(djourn - which repositions the cursor to the adjourn location in the file.

FIND COMMAND

The Find command offers several additional features in the Advanced Editor.

1. A <del> may be used to erase the characters backwards to the beginning of the pattern.

2. An <esc> may be used during an attempted find action to abort the search.

3. The response (a space) to the prompt '<target> not found...' may be typed ahead or an <esc> may be used.

4. Repeated backward finds of a target (using FS) do not require that the cursor be "manually" moved to the beginning of the target in order to find the second occurrence of the target.

5. Find patterns are preserved across files in any given editing session.

W(ORD COMMAND

This command is an additional command of the Advanced Editor that repositions the cursor (directional) to the first nonblank character of the next word. A "word" is defined as a sequence of characters not including a <space> or a <ret>.

<tab> COMMAND

The <tab> command repositions the cursor (directional) to the next user-specified tab stop. If the next tab stop is beyond the end or before the start of the printed line, the cursor is still positioned at the next tab stop.

In Insert mode, spaces are implicitly inserted from the initial cursor location to the tab stop.

Formatting Commands

Two Advanced Editor commands that effect formatting of text are different from the corresponding Screen-Oriented Editor commands.

A(DJUST COMMAND

The Adjust command prompt line shows the column-number location of the cursor as shown below by "xx":

```
------------------------------------------------------------------------
|>Adjust[xx]: L(just R(just C(enter <arrows> {<etx> to leave}          |
------------------------------------------------------------------------
```

The column number is displayed immediately; thus, A(djust <esc> may be used to show the current cursor location.

M(ARGIN COMMAND

In the Advanced Editor if an M is entered and Auto-indent and Filling are not set FALSE and TRUE, respectively, a prompt line appears as below:

```
------------------------------------------------------------------------
| Do you wish to margin this paragraph? (Y/N)                          |
------------------------------------------------------------------------
```

If a Y is entered, the current Auto-indent and Filling settings are suspended; the paragraph is margined (Auto-indent and Filling are temporarily set to FALSE and TRUE, respectively); and the original (saved) settings for Auto-indent and Filling are restored.

Text-Changing Commands

Some additional features are offered in three Advanced Editor commands. The additions to the R(eplace command are the same as those additions described in the Find command listed in Moving Commands in this section.

X(CHANGE COMMAND

The Exchange command allows characters to be exchanged on a one-for-one basis regardless of the initial cursor location within the line. That is, the Screen-Oriented Editor command does not allow the cursor to be moved left of the initial position; this command allows the entire line to be changed. When an end-of-line is encountered, the exchanged text is implicitly accepted, and the Insert command is automatically invoked.

C(OPY F(ILE COMMAND

This command offers the choice of copying "from" or "to" a file. Also, a list of marker choices is shown that allows copying (1) from a marker to the end of the file; (2) from the start of the file to the marker; or (3) between two markers. In the copy to a file, an additional choice is offered: copy the text between the current cursor location and the marker. The "copy to a file" option is a new feature of this command.

Control Commands

The Advanced Editor offers a new C(opy command option that copies editing controls to or from a file. Also, the S(et E(nvironment command contains additional features.

C(OPY C(ONTROLS COMMAND

This command copies the editing controls (for example, the standard set of controls '*ADV_ED.CONTROLS') from another file into the current edit buffer or to another file from the buffer.

The Advanced Editor S(et E(nvironment command prompt line is shown below:

```
---------------------------------------------------------------------
| Environ: <option letters>, S(et-tabs, '?', Q(uit                  |
---------------------------------------------------------------------
```

These choices (1) allow the editing environment to be changed; (2) allow tabs to be set; (3) allow interactive documentation to be viewed; and (4) allow a return to editing.

When compared with the Screen-Oriented Editor command, this command displays some additional information. The file name and status, <target patterns> (even complex or multiline ones), and tab positions are shown.

The S(et-tabs option causes the cursor to be placed on column Ø of the tab line as follows:

```
-------------------------------------------------------------------------
| Tab stops: Q(uit, <left,right>, S(et, R(eset, Z(ero, C(ol # Ø          |
-------------------------------------------------------------------------
```

The tab stops are shown on the display as below (T is tab stop and - is not):

```
T-------T-------T-------T-------T-------T-------T-------T-------T-------T------
```

The various S(et-tabs options are described in the following paragraphs.

Q(uit -- Terminates the tab setting operation.

<left,right> -- Moves the cursor over the tab line without changing the tabs. A repeat-factor may be used. The column number is displayed as the cursor moves.

S(et, R(eset -- Changes the tab at the current column and advances to the next tab stop (T). A 'T' is allowed for Set, and a - is allowed for Reset.

Z(ero -- Resets or clears all tabs from (and including) the current column to the end of the line.

C(ol -- Allows a new column to be entered directly and positions the cursor on the column # (at the end of the prompt) where the new column is entered.

| NOTE |

A column number outside the range Ø..79 is not allowed. For example, if the cursor is on column 1Ø, and an 11<- or 7Ø-> is entered, the cursor is not repositioned.

S(ET COMMAND

The *(macro and A(djourn options are enhancements to this command. The S(et *(macro allows macros (user-editing commands) to be defined. The S(et A(djourn option sets the explicit location to which the cursor is positioned on entry to the file.

V(ERIFY COMMAND

The Advanced Editor enhancement to this command allows the user to specify the redisplayed screen line number for the line to contain the cursor. That is, use of a repeat-factor enables the user to explicitly control the amount of text shown before and after the cursor on the redisplayed screen.

Q(UIT COMMAND

The Advanced Editor Q(uit command offers the user additional choices regarding the termination of an editing session. These options are divided into actions possible with the edit buffer and the next actions to be executed. These various choice are explained in detail in the Quit Command discussion in Section 3.2.1.

3.3 L2 EDITOR

The L2 Editor is a version of the Screen-Oriented Editor which allows editing of files which are too large to fit into the main memory buffer. This editor automatically produces a backup copy of the file being edited. Because the L2 Editor is an extended version of the Screen-Oriented Editor, very few differences exist between the two editors. These differences are described in the following subsections.

3.3.1 Initiating the L2 Editor

Unlike the Screen-Oriented Editor, the L2 Editor must be executed as a code file. That is, an X (for execute) is typed from the outer level command line of the III.Ø Operating System. A prompt asking which file to execute then appears. The response is "L2".

An alternate approach is to rename the L2 code file as the SYSTEM.EDITOR. In that case, the L2 Editor is called from the system main command line when an E is entered.

3.3.2 Space Constraints

If enough space does not exist on the disk to create the backup copy of the file, the L2 Editor issues the following message:

```
ERROR: Not enough room for backup!
```

To make enough space on the disk, either the Filer K(runch option (combines unused blocks at the end of the disk) must be used, or a file must be removed. Another disk could also be used.

Once sufficient disk space is available to create the backup copy, the L2 Editor displays the following message when executed:

```
 Copying to <filename>.back.
>Edit: A(djst C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap ?
 Reading. . .
```

3.3.3 Differences In Commands

Some of the L2 Editor commands are slightly different than the same Screen-Oriented Editor commands. These differences are pointed out in the following subsections.

J(mp (Jump Command)

The prompt that appears in response to the initiation of a Jump command is the same for both editors. However, the B(eginning and E(nd refer to the beginning and end of the buffer in the L2 Editor rather than referring to the file beginning and end as in the Screen-Oriented Editor.

F(ind (Find Command)

When a Find command is initiated, the L2 Editor displays "Finding...". If the pattern is not found in the contents of the buffer, the following prompt is displayed:

```
--------------------------------------------------------------------------
| End of buffer encountered.  Get more from disk?(y/n)                   |
--------------------------------------------------------------------------
```

If a Y for yes is entered, the L2 Editor moves another section of the file into the buffer and continues the search. The direction of the search still depends on the direction set.

S(et (Set Command)

The Set command functions the same in the L2 Editor as in the Screen-Oriented Editor except that 2Ø markers are allowed instead of 1Ø. Entering SM and SE cause the markers and the environment, respectively, to be set as in the Screen-Oriented Editor. However, the Environment status display contains some additional information for the L2 Editor. The following display shows the typical information shown in the Environment status display.

```
>Environment:  options <etx> or <sp> to leave
  A(uto Indent
  F(illing
  L(eft margin
  R(ight margin
  P(ara margin
  C(ommand ch
  S(et tabs
  T(oken def

  nnnn bytes used.  mmmm available.

There are n pages in the left stack, and m pages in
the right stack.  You have n pages of room, and at
most n pages worth in the buffer.

Markers:  <P1 P2 >P3
('<' indicates the marker is in the left stack, '>'
in the right stack, and no arrow indicates the marker
is in the current buffer)

Created mmddyy: Last updated mm yy dd(Revision n).
```

The S(et tabs option in the L2 Environment status display is accessed by typing an S while the display is on the screen; the following prompt appears.

```
Set tabs: <right,left,vectors> c(ol # N(o R(ight L(eft D(ecimal Stop <etx>
```

This option is not fully implemented; therefore, using R(ight, L(eft and D(ecimal has the same effect. That is, a variable tab stop is allowed rather than each tab being set eight characters apart.

NOTE

The environment information is not mutually compatible between the Screen-Oriented Editor and the L2 Editor. Either may be used on a file last updated by the other editor (subject to file size constraints); however, the environment information is reset to the default state.

Q(uit (Quit Command)

After all changes and additions are completed in the buffer being edited using the L2 Editor, a Q is entered to end the editing session. This process is the same as with the Screen-Oriented Editor except that the W(rite option is not available in the L2 Editor.

The other three options of the Q(uit command are slightly different than those of the Screen-Oriented Editor; these options are described below.

U(pdate - This option supplies additional information to indicate the file name and length. The information below is an example of the extra information given when a new file is created:

```
Writing.*
The workfile, X:F1.TEXT, is 73 blocks long.
The backupfile is X:F1.BACK.
```

The recently edited file is <filename>.TEXT and the original file with no changes is <filename>.BACK.

E(xit - This option prevents the <filename>.BACK from being created. The existing backup file is removed.

R(eturn - This option is the same as the R(eturn option of the Screen-Oriented Editor except that the cursor returns to the last editing change made in the buffer being edited.

3.3.4 L2 Additional Commands

The L2 Editor contains two commands that the Screen-Oriented Editor does not offer - the B(anish and N(ext commands.

B(anish (Banish Command)

The B(anish command moves characters from the buffer into the stack to allow more room in the buffer. This command is useful when an overflow condition would occur in completing a large insertion or copy. The left and right stacks are behind and ahead of the cursor, respectively. The screen is the boundary for the operation.

The B(anish command is initiated by typing a B; the following prompt line appears.

```
-------------------------------------------------------------------------
| >Banish:  To the L(eft or R(ight <esc>                                |
-------------------------------------------------------------------------
```

N(ext (Next Command)

The Next command is used to move beyond the bounds of the buffer. This command is initiated by entering an N; the following prompt appears.

```
---------------------------------------------------------------------------
| Next:  F(orwards, B(ackwards in the file: S(tart,                       |
|        E(nd of the file.  <esc>                                         |
---------------------------------------------------------------------------
```

When using an F or B, an implicit banish occurs using the cursor as the point of reference. If F is entered, everything above the screen is banished to the left stack. More characters are added to the bottom of the screen to extend the buffer in the forward direction.

If B is entered, the characters below the cursor are banished to the right stack and the lower part of the screen becomes blank. More characters are added above the screen.

The symbolic file is depicted below.

```
-----------------------------------------------
| LEFT STACK    |               | RIGHT STACK  |
|               |    BUFFER     |              |
| START         |               | END          |
-----------------------------------------------
```

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

3.4 LINE-ORIENTED EDITOR (YALOE)

The Line-Oriented Text Editor, YALOE, is designed for use in systems having a teleprinter or teletypewriter as the system console. This editor is useful for creating a GOTOXY procedure in the case where the CRT to be used with the system is not compatible with the system as shipped. Because the screen editors are dependent on a correct GOTOXY procedure, YALOE is used to create the procedure. Once the GOTOXY procedure is bound in and the SYSTEM.MISCINFO file created, the screen editors can be used.

YALOE provides facilities for the following actions:

- Listing lines of text from the work file (the file being created or modified).
- Transferring text between the text buffer and files.
- Relocating the cursor (the current position in the text being manipulated).
- Inserting, deleting, replacing, and exchanging text.

YALOE also provides a macro facility, allowing the user to execute a frequently used group of commands by issuing a single command.

3.4.1 General Information

The YALOE Text Editor is designed for use in systems that have a teleprinter or teletypewriter as the system console rather than a video display terminal. YALOE must be executed from the system main command line or else must be renamed SYSTEM.EDITOR.

YALOE assumes the existence of a work file but is not dependent on the work file being present. The work file can be created after entering YALOE. If a work file already exists, YALOE prints the following message.

```
| Workfile STUFF read in.                                          |
```

If YALOE is called and the work file is empty, the following message appears:

```
| No workfile read in.                                             |
```

YALOE operates in either Command or Text Mode and is in Command Mode when entered. In Command Mode, all keyboard input is assumed to be commands. Each command may be terminated by <esc>. The commands may be strung together. No commands in a string (or singly) are executed until the final command in the string is followed by <esc> <esc>. Spaces, carriage returns and tabs within a command string are ignored unless they appear in a text string. When the execution of a command string is complete, YALOE prompts for the next command with an asterisk (*). In contrast to other levels of the III.Ø Operating System, a prompt line of available commands is not given.

If an error is encountered during command execution, the command is terminated at that point without completing execution.

Text Mode is entered whenever a command is typed that must be followed by text. All succeeding characters are considered to be text until the next <esc>. The commands that require text are F(ind, G(et, I(nsert, M(acro define, R(ead file, W(rite to file, and eX(change.

NOTE

> When typed, <esc> echoes a dollar sign ($). The <esc> terminates each text string and causes YALOE to reenter the Command Mode. A double <esc> terminates the command string and causes YALOE to begin execution.

The work file is stored in the text buffer. This area is allocated dynamically by the ? command. YALOE works only with files that fit completely within the text buffer.

The cursor is the position in the file where the next command is to be executed. Most edit commands function in relation to the cursor.

Some of the YALOE commands described here require a command argument to precede the command letter. Usually, the argument specifies the number of times the command should be performed or the particular portion of text to be affected by the command. With some commands, the specifications are implicit and no argument is needed. The command arguments used by YALOE are as follows:

n   Any integer, signed or unsigned. Unsigned integers are assumed to be positive. In a command that accepts an argument, the absence of the argument implies 1 (only one execution) or minus 1 if only the minus sign is present.

m   A number in the range Ø through 9.

o   The beginning of the current line.

/ The same as 32700. A "-/" is -32700. Used for a large repeat factor.

= Represents -n where n equals the length of the last text argument used. Applies only to the J, D, and C commands.

3.4.2 Special Key Commands

Various keys on the keyboard have special functions, some of which are unique to YALOE. These commands are described below. Those control keys that do not appear below are ignored and discarded by YALOE.

<esc>

The escape key is echoed as a dollar sign ($) on the console. A single <esc> terminates a text string. A double <esc> executes a command string.

RUBOUT
<linedel>

On hard-copy terminals, line delete is echoed as "<ZAP" and a carriage return. On others, it clears the current line on the screen. In both cases, the contents of that line are discarded by YALOE.

CTRL H
<chardel>

On hard-copy terminals, character delete is echoed as a percent sign (%) followed by the character deleted. Deletions are right to left, with each character deleted, erased by the %, up to the beginning of the command string. CTRL H may be used in both Command and Text Modes.

CTRL X

CTRL X causes YALOE to ignore the entire command string and respond with a carriage return and an asterisk (*) to prompt for another command. The command string being ignored may be on several lines. All lines back to the previous * prompt are ignored. (A character delete is confined to one line.)

CTRL O

CTRL O causes YALOE to switch to the optional character set (bit 7 turned on). This command argument applies only to the TERAK 8510A terminal.

|NOTE|

If strange characters appear on the terminal, CRTL O may have been hit accidentally. This condition is corrected by again typing CTRL O.

CTRL F
flush

CTRL F causes YALOE to discard all output to the terminal until the next CTRL F is typed.

CTRL S
stop

CTRL S causes YALOE to store all output to the terminal until the next CTRL S is typed.

3.4.3 Input/Output Commands

The commands that control I/O are described below.

LIST

The LIST command is specified by typing L for L(ist. This command causes YALOE to print a specified number of lines on the terminal without moving the cursor. Variations of this command are explained below.

*-3L$$ Prints all characters starting at the third preceding line and ending at the cursor.

*5L$$ Prints all characters beginning at the cursor and terminating at the fifth carriage return (line).

*ØL$$ Prints from the beginning of the current line up to the cursor.

VERIFY

The VERIFY command is specified by typing V for V(erify. This command causes YALOE to print the current text line on the terminal. The position of the cursor within the line has no effect on the command, and the cursor is not moved. No arguments are used. VERIFY is equivalent in effect to a *ØL$$ list command.

WRITE

The WRITE command is specified by typing W for W(rite followed by the file title, in the following format:

```
*W<file title>$
```

The file title is any legal file title, except that the file type is not given. YALOE automatically appends ".TEXT" as a suffix unless the title ends with a ".", "]", or ".TEXT". If the title ends in a ".", the period is stripped from it.

The WRITE command writes the entire text buffer to a file having the given file title. The cursor is not moved, nor are the contents of the text file altered. If the volume specified by the file title has insufficient room for the text buffer, the following error message appears:

```
---------------------------------------------------------------------
| OUTPUT ERROR. HELP!                                                |
---------------------------------------------------------------------
```

The text buffer can be written to another volume.

READ

The READ command is specified by typing R for R(ead followed by the file title, in the following format:

```
*R<file title>$
```

YALOE attempts to locate the file title as given. If no file is found having that title, a ".TEXT" is appended and a new search is made.

The READ command inserts the specified file into the text buffer, starting at the location of the cursor. If the file read does not fit, the entire text buffer is undefined in content. This situation is an unrecoverable error.

QUIT

The QUIT command is specified by typing Q for Q(uit and has several forms, as follows:

QU   Quit and update by writing out a new SYSTEM.WRK.TEXT.
QE   Quit and escape YALOE; do not alter the work file.
QR   Do not quit; return to YALOE.

If Q is typed alone, a prompt is sent to the terminal giving the above choices. An option must be entered (U, E, or R).

The QU command is a special case of the WRITE command. If QU does not work, W can be used to write out SYSTEM.WRK.TEXT followed by QE to exit from YALOE. QR is used to return to YALOE after a Q has been typed accidentally.

ERASE

The ERASE command is specified by typing E for E(rase. This command functions only with video display terminals and causes YALOE to erase the screen.

O

The O command is specified by typing O. This command functions only with video display terminals and causes YALOE to display the text around the cursor each time the cursor is moved. The argument for the O command specifies the number of lines to be displayed. This option is in a disabled state when YALOE is entered. If needed, the option must be enabled by using the O command. A second O disables the option. The location of the cursor is denoted by a split in the line of text.

3.4.4 Moving Commands

The moving commands relocate the cursor to a new position. These commands are important because most other editing commands are dependent on cursor positioning. The moving commands are described below.

The direction of cursor movement is specified in the commands by the sign of the argument. A positive (+n) argument gives the number of characters or lines to move in a forward direction; and a negative argument (-n), in a backwards direction.

Carriage return characters are treated the same as any other character in text except that the <cr> denotes the end of a line of text.

Examples of the moving commands are given in Figure 3-9. In the examples, the cursor position is indicated by an up arrow (^) although the cursor does not actually appear on the teleprinter or teletypewriter.

JUMP

The JUMP command is specified by typing J for J(ump. JUMP moves the cursor a specified number of characters in the text buffer. Movement may be either forward or backward and is not restricted to the current line.

ADVANCE

The ADVANCE command is specified by typing A for A(dvance. ADVANCE moves the cursor a specified number of lines. The cursor is then positioned at the beginning of the line to which it moved. An argument of zero moves the cursor to the beginning of the current line. Movement may be either forward or backward.

{Here are the original lines and the cursor position.}

THE TIME HAS COME<cr>

THE WALRUS SAID<cr>

TO TALK OF MANY THINGS<cr>

Example 1. *8J$$ moves the cursor forward eight characters to the next line between the K and the space.

TO TALK OF MANY THINGS<cr>

Example 2. *-2A$$ moves the cursor to the beginning of the second preceding line.

THE TIME HAS COME<cr>

Example 3. *BGTWINE$=J$$ moves the cursor to the beginning of the text buffer, then starts searching for the string "TWINE". When the string is found, the cursor is positioned immediately before it.

Figure 3-9. Example of Moving Commands.

BEGINNING

The BEGINNING command is specified by typing a B for B(eginning. BEGINNING moves the cursor to the beginning of the text buffer.

GET and FIND

The search commands GET and FIND are synonymous. GET is specified by typing G and FIND by typing F. With either command, the current text buffer is searched starting from the location of the cursor for the nth occurrence of a specified text string. On completion of a successful search, the cursor is positioned immediately following the nth occurrence if n is positive and immediately before, if n is negative. If the search is unsuccessful, YALOE generates an error message, and the cursor is positioned at the end of the buffer if n is positive and at the beginning if n is negative.

3.4.5 Text Changing Commands

The text-changing commands add to, remove, or change the text. These commands are described in the following paragraphs; examples are given in Figure 3-10.

INSERT

The INSERT command is specified by typing I for I(nsert. INSERT causes YALOE to enter Text Mode to add characters immediately following the cursor until an <esc> is typed. After insertion is completed, the cursor is positioned immediately following the last character inserted.

Occasionally, with a large insertion, the temporary buffer becomes full. Before this situation occurs, the following message is printed on the console.

```
| Please finish.                                                       |
```

Typing <esc> <esc> terminates the insertion at that point so that the temporary buffer can be emptied into the text buffer. Insertion can then be continued by again typing I to reenter Text Mode. Not typing I causes the characters that are next entered as insertions to be interpreted as commands.

*-4D$$	Deletes the four characters immediately preceding the cursor, even if they are on the previous line.
*B$GTWINE $=D$$	Moves the cursor to the beginning of the text buffer, searches for the string "TWINE", and deletes it.
*/K$$	Deletes all lines in the text buffer from the line in which the cursor is positioned to the end of the buffer.
*OCAAA$$	Replaces the characters from the beginning of the line to the cursor with "AAA" (same as *OXAAA$$).
*BGA$=CB$$	Searches for the first occurrence of "A" and replaces it with "B".
*-3XNEW$$	Exchanges all characters beginning with the first character on the third line back and ending at the cursor with the string "NEW".

Figure 3-10. Examples of Text-Changing Commands.

DELETE

The DELETE command is specified by typing D for D(elete. DELETE removes a specified number of characters from the text buffer, starting with the position of the cursor. On completion of the deletion, the cursor is positioned immediately following the deleted text.

KILL

The KILL command is specified by typing K for K(ill. KILL deletes a specified number of lines from the text buffer starting at the position of the cursor. On completion, the cursor is positioned at the beginning of the line following the deleted text.

CHANGE

The CHANGE command is specified by typing C for C(hange. CHANGE replaces n characters, starting at the position of the cursor, with the given text string. On completion, the cursor is positioned immediately following the changed text.

EXCHANGE

The EXCHANGE command is specified by typing X for eX(change. EXCHANGE exchanges n lines, starting with the line on which the cursor is located, with the indicated text string. The cursor remains at the end of the changed text on completion of the command.

3.4.6 Miscellaneous Commands

Some YALOE commands do not fall into a category but are miscellaneous commands for various purposes. These commands are described in the following paragraphs.

SAVE

The SAVE command is specified by typing S for S(ave. SAVE copies the specified number of lines into the save buffer, starting at the cursor. On completion, the cursor position is unchanged, and the contents of the text buffer are unaltered. Each time SAVE is executed, the previous contents of the save buffer, if any, are destroyed. If executing a SAVE will cause the text buffer to overflow, YALOE generates a message and does not perform SAVE.

UNSAVE

The UNSAVE command is specified by typing U for U(nsave. UNSAVE inserts the entire contents of the save buffer into the text buffer at the cursor. On completion, the cursor is still positioned before the inserted text. If the text buffer does not have enough room for the contents of the save buffer, YALOE generates a message to this effect and does not execute UNSAVE.

The save buffer may be removed by typing OU.

MACRO

A macro is a single command that performs a string of standard but related commands. Any group of frequently used commands can be grouped into a macro to eliminate the need for having to write the whole set of instructions whenever they are needed. The user may create macros by using the M(acro command. The MACRO command is specified by typing M for M(acro in the following format:

```
mM%command string%
```

In this format, m is an integer in the range Ø through 9. MACRO is used to define a maximum of ten macros. The default number is 1. The command string is stored in the macro buffer m. The command string delimiter (% in the above case) is always the first character following the M. The delimiter may be any character that does not appear in the macro command string itself. The second occurrence of the delimiter terminates the macro.

All characters except the delimiter are legal macro command string characters, including a single <esc>. All YALOE commands are legal. An example of a macro is given in Figure 3-11.

If an error occurs when defining a macro, the following error message is generated:

Error in macro definition.

The macro must be redefined.

```
*4M%FPREFACE$=CEND PREFACE$V$%$$
```

This example defines macro number 4. When macro 4 is executed, YALOE looks for the string "PREFACE", changes it to "END PREFACE", and then displays the change to verify it.

Figure 3-11. Example of a Macro.

N (Execute Macro)

The N command, which executes a specified macro command string, is specified by N in the following format:

```
nNm$
```

The n is simply any command argument (for example, a repeat factor), and m is the macro number to be executed. If m is omitted, one is assumed. Because m is technically a command text string, the N command must be terminated by <esc> (echoed as $).

Attempts to execute undefined macros result in the generation of the following error message:

```
Unhappy macnum.
```

Errors encountered during macro execution generate the following error message.

```
Error in macro.
```

? (List)

The ? command is specified by typing ?; this command prints a list of all commands, the sizes of the text buffer, the save buffer, and the memory still available for expansion.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

4. THE FILE HANDLER (FILER)

The File Handler (Filer) is a separate compartment of the III.Ø Operating System which handles, identifies, structures, and restructures files used on the system. The Filer offers commands that provide means to keep track of files, to manipulate files, and to maintain files and diskettes/disks.

That is, the Filer commands can be generally categorized. These categories are (1) information commands to provide lists of files and volumes; (2) manipulative commands to handle the system work file; and (3) disk and file maintenance commands to allow the following operations:

- Moving files and directories.
- Copying files and volumes.
- Creating files, changing file names, and removing files.
- Checking disks/diskettes for corrupted or damaged areas.
- Creating new directories so that information can be recorded.
- Changing the system date so that updated files reflect a current date.
- Changing the default volume on the system.

Refer to Section 1.3.2 for a brief description of the Filer commands. This chapter contains a detailed explanation of the Filer commands in Section 4.2. Section 4.1 presents some general information regarding the Filer -- namely, accessing the Filer (4.1.1); Files, Volumes and File Specifications (4.1.2); and the Filer Command Categories (4.1.3).

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

4.1 GENERAL INFORMATION

Generally, the Filer manipulates and maintains files, which are the basic unit of permanent storage used with the III.Ø Operating System. Some Filer functions relate to files stored on disks/diskettes; other functions relate to unblocked device files such as a printer or console file.

4.1.1 Accessing the Filer

The Filer is accessed by typing F (for F(iler) from the system outer level command line. In response to the F, the following Filer main command line is displayed across the top of the screen.

```
| Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit   |
```

This command line lists some of the Filer commands; to display the secondary Filer command line, a ? is entered. The secondary command line is shown below:

```
| Filer: B(ad-blk, E(xt-dir, K(runch, M(ake, P(refix, V(ols, X(amine, Z(ero   |
```

The commands listed in the secondary command line can be accessed directly from the Filer main command line by typing the first character of the command or can be accessed from the secondary command line after it is displayed in response to the entry of a ?.

All Filer commands are initiated by typing the first character of the command on the console. Many of the commands display additional prompt lines in order to have the information necessary for execution. Answering a "Yes/No" question on a prompt line with any character other than a Y or y constitutes a No answer. In most cases, typing an <esc> returns control to the Filer main command line.

4.1.2 Files, Volumes, and File Specifications

Refer to Section 2.1 for an explanation of files and to Section 2.2 for an explanation of volumes. These subjects are important to understand regarding the Filer.

Another important subject in regards to the Filer is that of file specifications. Many Filer commands require a file specification. Figure 4-1 illustrates the syntax of a file specification.

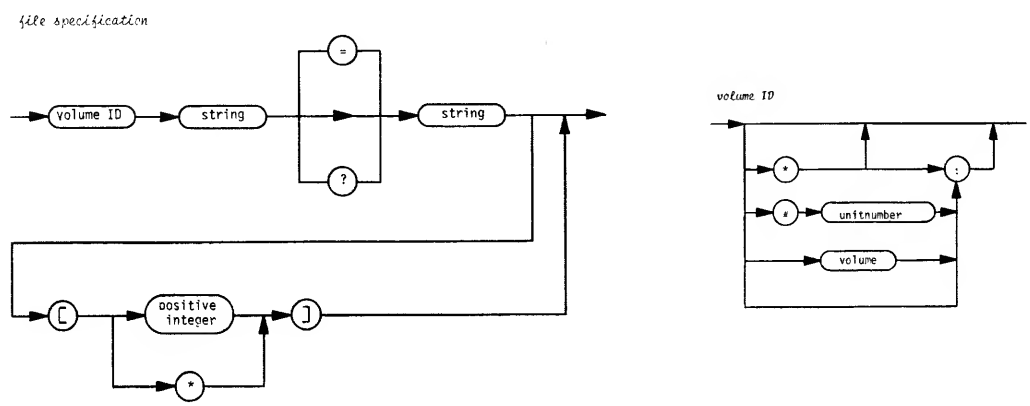


Figure 4-1. Syntax for a File Specification.

Whenever a file name is requested, as many files as desired may be listed. The file names must be separated by commas, and the list must be terminated with a carriage return (<ret>). Commands that operate on single file names continue reading the names from the list and operate on each until no names remain. Commands that operate on two file names at once (for example, CHANGE and TRANSFER) continue reading names in pairs until one or no file names remain. If only one file name remains in the above case, the Filer prompts for the second name. If an error is found in the list, the entire list is flushed. The rules for legal file names are listed in Section 2.1.

The Filer performs the requested action on all files meeting the file specifications. Some specifications are made by using wild card characters. The wild card characters "=" and "?" specify a subset of a directory. For example, a file specification that contains "PUB=TEXT" as a string to specify a subset causes the Filer to perform the requested action on all files whose names begin with the string "PUB" and end with the string "TEXT".

If a ? is used in place of the =, the Filer requests verification before performing the requested action. Generally, the ? causes the Filer to request verification before completing any command. Using the ? alone causes the Filer to act on every file in a volume directory and to request verification for each file before completing the command for that file. For example, the ? can be used in file transferring from one media (or diskette) to another to prompt the user regarding the transfer of each file.

In using wild card characters, either or both strings may be empty. For example, a subset specification "=<string" or "<string>=" or even "=" is valid. In the case where both strings are empty, the Filer acts on every file in the volume directory.

In some contexts, the pattern '[number]' at the end of a file name is interpreted as a block size specification and is not part of the actual file name.

4.1.3 Filer Command Categories

The Filer commands can be grouped into three main categories as follows:

- Information Commands
- Manipulative Commands for the System Work File
- File/Disk Maintenance Commands

These categories group the commands by general function. A list of the commands in each group is presented in Table 4-1.

Table 4-1. Filer Commands By Category.

Information	Work File Manipulation	File/Disk Maintenance	
L(dir	G(et	R(em	K(runch
E(xt-dir	S(ave	C(hng	M(ake
V(ols	W(hat	T(rans	P(refix
	N(ew	D(ate	X(amine
		B(ad-blks	Z(ero

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

4.2 FILER COMMANDS

Although the Filer commands can be grouped into functional categories, the descriptions of the commands presented in this section are ordered based on their placement in the Filer main and secondary command lines. The commands listed on the main command line are presented first (in left to right order as they appear on the prompt line). Then, the secondary commands are described based on their order in the secondary prompt line.

4.2.1 G(et (Get Command)

The Get command is used to load a specified file into the work area as the work file. The Get command is initiated by typing G from the Filer main command line. If no files named SYSTEM.WRK.TEXT or SYSTEM.WRK.CODE exist in the directory, the Filer responds with the following prompt:

```
| Get what file?                                                    |
```

If either or both the system work files exist, the Filer asks the following question:

```
| Throw away current workfile?                                      |
```

If the response is yes, that file (or files) is removed from the disk. If the response is other than yes, the Get command action is aborted and the Filer main command line reappears.

In response to the first prompt, the file name entered is loaded as the work file. The suffixes .TEXT and .CODE are not required. If a text and code file exist for the file name entered, both are loaded. If one or the other type of file exists, that file is entered although neither .TEXT nor .CODE were specified. Also, the entire file specification is not required. If the volume ID or name is not given, the default disk is assumed. Wild card characters are not allowed, and the size specification is ignored.

When the Filer completes the loading operation, one of the following messages is printed -- depending on the files that exist on the disk.

- Text and code file loaded.
- Code file loaded.
- Text file loaded.

If no file exists with the specified name, the Filer responds:

```
| No file loaded.                                                    |
```

An example of the Get command is presented in Figure 4-2. In the figure, the characters entered by the user are shaded; comments are enclosed in braces ({}).

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

G

Get what file? #5:POP

{Two files named POP.TEXT and POP.CODE exist on device #5.}

Text and code file loaded.
```

Figure 4-2. Get Command Example.

4.2.2 S(ave (Save Command)

The Save command is used to save (write to disk) the work file. Both components (SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE) of the work file are saved (1) under the original file name if a Get command was used or (2) under a different file name as specified by the user. The Save command is initiated by typing S from the Filer main command line.

If the work file was created by the Get command, the Filer prompts as below:

```
| Save as <file name>?                                               |
```

If a yes response is given and the file already exists, the Filer prompts:

```
| <file name> exists...remove it?                                    |
```

If a yes response is entered, and the original file is located on other than the default volume, the following message appears:

```
|<vol ID>:SYSTEM.WRK.TEXT transferred to <file name>                  |
```

In this case, the SYSTEM.WRK.TEXT (or .CODE) file remains on the system volume until the work file is cleared. The work file can be cleared by a Get or New command.

If the original file is on the system disk or default volume, the message regarding transferring the file does not appear. The original file is updated if the work file is saved with the name of the original file. In this case, the "SYSTEM.WRK" files disappear when the Save command is used to write the work file to the original file name or to a new file name.

The suffixes ".TEXT" and "CODE" are not required when using the Save command. The III.Ø Operating System automatically appends to correct suffix.

If the volume name is not specified, the default volume is assumed. Wild cards are not allowed, and the size specification does not apply.

Figures 4-3 and 4-4 give examples of the Save command. Comments are enclosed in braces ({}), and user input is shaded.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

  G
  Get what file? SFW1:TEST <ret>
  Text and Code file loaded.
  Q
```

{Through the above sequence of commands, the file "TEST" on the volume named "SFW1" is made the work file. This file has a code file associated with it. After the Editor is used to load the file into memory and to make changes to the file and the code is recompiled using the Run command, the file is to be saved as "TEST1". The following sequence of commands is used to save the work files.}

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit
  S
  Save as SFW1:TEST? N
  Save as what file? SFW1:TEST1 <ret>

  SYS1:SYSTEM.WRK.TEXT transferred to SFW1:TEST1.TEXT

  SYS1:SYSTEM.WRK.CODE transferred to SFW1:TEST1.CODE
```

{The updated versions of the file are, thus, transferred to the volume "SFW1". The "SYSTEM.WRK" and "SYSTEM.CODE" files remain on the default volume until the work space is cleared.}

Figure 4-3. Example of the Save Command Across Volumes.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

  {A text file is created using the Screen-Oriented Editor, and the work file
  is updated using the U option of the Editor Q(uit command.  Thus, the file
  is written on the default volume as SYSTEM.WRK.TEXT.  The work file is
  temporary and, at that point, the work file is not saved.  The following
  sequence of commands saves the file as PLAY.TEXT on the default volume and,
  at the same time, removes the SYSTEM.WRK.TEXT file.}

S
Save as what file? PLAY <ret>
TEXT file saved
```

Figure 4-4. Example of Save Command With SYSTEM.WRK File.

4.2.3 W(hat (What Command)

The What command displays the name and status (saved or not saved) of the current work file. This command is initiated by typing W from the Filer main command line.

If SYSTEM.WRK.TEXT and/or SYSTEM.WRK.CODE exist on the default volume but a Get command was not used to create a named work file, the following response to a What command appears.

```
Workfile is not named (not saved)
```

However, if the Get command is used to load a named work file (for example, SFW1:PLAY.TEXT), and the file is edited and updated but not saved, the response to the What command is as follows.

```
Workfile is SFW1:PLAY (not saved)
```

If neither named nor unnamed work files are present, the following response to the What command appears.

```
No workfile
```

4.2.4 N(ew (New Command)

The New command clears the work space so a new work file can be created. The New command removes any work files on the system volume so that no default file exists to be used automatically by the E(dit, C(ompile, or R(un commands. All versions of the work file (SYSTEM.WRK.TEXT and/or SYSTEM.WRK.CODE) are removed from the system directory by the New command.

The New command is initiated by typing N from the Filer main command line.

If a system work file exists at the time a New command is executed, the following prompt appears:

```
| Throw away current workfile?                                    |
```

If a Y is entered, the work space is cleared. If an N is entered, the main command line of the Filer is redisplayed.

If a backup work file exists (as created when the L2 Editor is used to create the work file), the following prompt appears:

```
| Remove <workfile name>.Back?                                    |
```

4.2.5 L(dir (List Directory Command)

The List Directory command gives information pertaining to the specified directory of a selected disk/diskette volume. All or part of the directory is displayed (default destination is CONSOLE:) as specified. The List Directory command is initiated by typing L from the Filer main command line. The following prompt appears:

```
| Dir listing of what vol ?                                        |
```

The directory can be listed to the volume and file specified. The default volume is "CONSOLE:", but the listing can be directed to a file on disk, "PRINTER:", or "REMOTE:". The file specification, in this case, must be in terms of source and destination.

The source file specification consists of a mandatory volume name where ":" indicates the prefixed volume and an optional file name, which may include subset-specifying strings. If subset-specifying strings are used, a wild card is used. The source information must be separated from the destination information (if given) by a comma.

When entered, the destination specification includes the volume name and, if the volume is block-structured, a file name. The file size is ignored.

Usually, this command is used to list the entire directory. The directory listing that appears on the screen fills the screen, stops, and prompts as below to continue viewing the listing.

```
---------------------------------------------------------------------------
| Type <space> to continue                                                |
---------------------------------------------------------------------------
```

When the space bar is pressed, the next screen of information is displayed until the directory list is completed. The directory is limited to 77 file entries. (See 2.2 regarding this limitation.)

Figures 4-5 and 4-6 give examples of the L(dir command. Figure 4-5 presents an example of this command as used to print the directory on a serial printer. Figure 4-6 shows a directory listed by this command to the console. Comments are enclosed in braces ({}); user input is shaded.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

L
Dir listing of what vol? H30S:,REMOTE: <ret>

   {This specification causes the directory for the system diskette to be
   printed on the serial printer}
```

Figure 4-5. Example of List Directory Command (List to Serial Printer).

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

 L
 Dir listing of what vol? #4 <ret>

   {The following directory of the system diskette is displayed on the screen;
   the volume name is H3OS.}

 Type <space> to continue

 H3OS:
 SYSTEM.COMPILER    79    4-Feb-82
 LIBRARY.CODE       23    4-Feb-82
 DISASM.CODE        24    4-Feb-82
 SYSTEM.LINKER      27   25-Feb-82
 SYSTEM.MISCINFO     1    8-Mar-82
 BINDER.CODE         8    4-Mar-82
 LIBMAP.CODE         9    4-Feb-82
 SYSTEM.EDITOR      52   29-Jul-81
 ADV.EDITOR         79   23-Mar-82
 CONFIGURE.CODE     25   25-Mar-82
 WFORMAT.CODE       18   24-Mar-82
 MARKDUPDIR.CODE     5   23-Mar-82
 COPYDUPDIR.CODE     5   23-Mar-82
 BOOTER.CODE         5   23-Mar-82
 SETUP.CODE         39   31-Mar-82
 PATCH.CODE          8    2-Apr-82
 SYSTEM.LIBRARY     30    7-Apr-82
 COPY.CODE           5   25-Mar-82
 SYSTEM.PACAL      110   25-Mar-82
 SYSTEM.FILER       48   19-Apr-82
 FORMAT.CODE        14    2-Apr-82
 BOOTMAKE.CODE       7    2-Apr-82

 <space>  {After the space bar is pressed, the following information is
          displayed.}

H3OS:
BOOT.CODE           10    2-Apr-82
23/23 files<listed/in-dir>, 632 blocks used, 346 unused
```

Figure 4-6. Example of List Directory Command (List to Console).

In Figure 4-6, the file names are listed that are contained in the directory for H3OS (column one). Column two gives the number of blocks in the file; column three is the date the file was last written. This date could be the creation date, if the file has not been "written to" since that date. This date is changed each time the file is written; the date is based on the date set through the Filer Date command.

The bottom line of the directory listing shows how many file names are shown and the total of the file names in the directory. In the example, 23 file names are listed out of a total of 23 file names in the directory. However, if a subset-specifying string, for example, "#4:SYSTEM.=", had been entered, seven out of 23 files would be shown (7/23). Of the total blocks on the diskette, 632 blocks are used and 346 remain available for use.

4.2.6 R(em (Remove Command)

The Remove command is used to remove file names from the disk directory, leaving the space formerly occupied by the file marked as unused. This command changes the directory; the information in the removed file still resides on the disk/diskette. However, once the file name is removed from the directory, the file information is no longer accessible to the user. The III.Ø Operating System now considers the area of the disk on which the file is written to be free space. Other files may now write to that space.

The Remove command is executed by typing R from the Filer main command line. The following prompt appears in response to this command.

```
| Remove what file?                                                   |
```

The Remove command requires one file specification for a file to be removed. The following rules apply in response to this prompt.

- The volume name or device number is required unless the file resides on the default disk. A colon is required to separate the volume identification from the file name. For example, SFW1:test1.CODE, where "SFW:" is the volume identification and "test1.CODE" is the file name.

- The file name extension is required. That is, the ".TEXT" or ".CODE" suffix must be included as part of the file name.

- Wild cards are permitted as described below:

  - A file name consisting of a single letter followed by an equal sign (=) instructs the Filer to remove all files beginning with that letter. The equal sign may also be used to remove groups of files with common letters either at the beginning or end of the file name.

  - A file name that consists solely of an equal sign causes every file in the directory to be removed.

  - The use of a question mark (?) causes a prompt for confirmation to appear before each file is removed. The question mark may be substituted in either of the above wild card specifications.

A list of files may be removed by entering the file names separated by commas.

The Filer prompts for confirmation whether or not to remove the file name from the directory; the prompt is shown below.

```
------------------------------------------------------------------------------
| Update directory?                                                          |
------------------------------------------------------------------------------
```

A Y or y response causes the Filer to remove the file from the directory; any other response leaves the directory in its original state. In either case, the execution of the Remove command is complete, and the Filer main command line is redisplayed.

|NOTE|

The Remove command should NOT be used to remove the SYSTEM.WRK.TEXT or SYSTEM.WRK.CODE files. These files should only be removed through the New or Get commands.

Because the "SYSTEM.WRK" files are referenced in an operating system table, even if the files are removed by the R(emove command, the III.Ø Operating System still lists the files as being present. This situation provides error messages such as "Workfile lost".

Figures 4-7, 4-8, and 4-9, give examples of the Remove command. Figure 4-7 illustrates the question mark to cause prompting for each removal; Figure 4-8 illustrates the use of the wild card equal sign; and Figure 4-9 illustrates removal of multiple files separated by commas. Comments are enclosed in braces ({}); user input is shaded.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

  {The following commands and responses remove all files that begin with an
  "a" as selected by the user.  The ? is used to cause the filer to prompt
  for each file to be removed.}

R
Remove what file? QSTGS:a? <ret>
Remove ASDIOE? N
Remove ASDIOE.MASK? N
Remove APUNIT.CODE? N
Remove ABC.TEST.TEXT? N
Remove ALPHA.TEXT? Y
Remove ADD3.TEXT? Y
Update directory? Y
```

Figure 4-7. Remove Command Example Using Wild Card (?).

In Figure 4-7, the user confirms or denies the removal of file names that begin with an "A". After Y is entered in response to the "Update directory?" prompt, the Filer main command line reappears.

```
Filer: G(et, S(ave, What, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

   {The following commands and responses remove all files which begin with
   "NEW" and all files which end with "TEST". The equal sign is used as a
   wild card to effect the removals.}

R
Remove what file? QSTGS:NEW=<ret>
QSTGS:NEW/TEST1.TEXT      removed
QSTGS:NEW/TEST2.TEXT      removed
QSTGS:NEW/TEST3.TEXT      removed
Update directory? Y

   {The Filer main command line reappears after the directory update.}

Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

R
Remove what file? QSTGS:=.TEST.TEXT <ret>
QSTGS:BCD.TEST.TEXT       removed
QSTGS:CDE.TEST.TEXT       removed
Update directory? Y
```

Figure 4-8. Remove Command Examples Using Wild Card (=).

Figure 4-8 above illustrates the removal of file names that have common beginning or ending nodes by using the equal sign as a wild card.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

   {The following commands and responses remove a series of three files.}

R
Remove what file?  QSTGS:HDW2.TEXT,QSTGS:SFW1.TEXT,QSTGS:SFW2.TEXT <ret>
QSTGS:HDW2.TEXT        removed
Update directory?  Y
QSTGS:SFW1.TEXT        removed
Update directory?  Y
QSTGS:SFW2.TEXT?       removed
Update directory?  Y
```

Figure 4-9. Remove Command Example - File Sequence.

4.2.7 C(hng (Change Command)

The Change command changes a file name or volume name. This command is initiated by typing C from the Filer main command line. After typing C, the Filer prompts for the file to be changed as shown below.

```
| Change what file?                                                     |
```

This command requires two file specifications: (1) the file or volume name to be changed and (2) the new name. These specifications may be entered on one line in response to the prompt separated by a comma, or they may be entered on two lines with a return (<ret>) separating the first from the second. In the case where the specifications are separated by a <ret>, the following second prompt appears.

```
| Change to what?                                                       |
```

Any volume name or device number in the second specification is ignored because the Filer recognizes that the file is on the same volume (or is the same volume, when the volume name is changed). The size specification, if given, if also ignored.

Wild card specifications are permitted. That is, the portion of the original file name represented by the equal sign is duplicated in place of the equal sign in the new file name. If a wild card is used in the first specification, it must also be used in the second.

Any subset-specifying strings used in the first specification are replaced by the analogous strings (replacement strings) in the second. That is, string characters may be placed before or after the equal sign, or both, in the first

or second file specification. If the equal sign is used alone as a subset-specifying string (both strings are empty), the Filer considers the specification to apply to all files in the directory.

The file name suffixes ".TEXT" and ".CODE" must be given as part of the file specification. Also, the Filer does not change any name if the new name exceeds 15 characters in length. When using a subset-specifying string to change the names of a group of files, if one of the new file names will exceed 15 characters, that file name is not changed. If all the new file names will exceed 15 characters, none of the changes are made.

To change a volume name, the volume name followed by a colon must be specified for both the old and new names. No reference to files in the directory should be made.

Figures 4-10, 4-11, 4-12 and 4-13 give examples of the Change command. Figure 4-10 illustrates the separation of the file specifications by a <ret>; Figure 4-11 and 4-12 give examples of wild card specifications; and Figure 4-13 is an example of a volume name change. In the figures, comments are enclosed in braces ({}), and user input is shaded.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit


C
Change what file? QSTGS:JUNK.TEXT <ret>
Change to what? PLAY.TEXT
QSTGS:JUNK.TEXT changed to PLAY.TEXT
```

{The above command and responses effect the file name change.
The volume name is not repeated because the file is assumed
to be on the same volume.}

Figure 4-10. Two-Line File Name Change.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit
```

{An example using a wild card specification is given below. The commands and responses below show the change of three file names. The wild card specifications change the subgroups of files that begin with "SFW" and end in "XT" to file names beginning with "OLDSFW" and ending in "XT". The original file names in the directory are: SFW1.TEXT,SFW2.TEXT,HDW1.TEXT, HDW2.TEXT, and SFW3.TEXT.}

```
C
Change what file? QSTGS:SFW=XT,OLDSFW=XT <ret>
QSTGS:SFW1.TEXT        changed to OLDSFW1.TEXT
QSTGS:SFW2.TEXT        changed to OLDSFW2.TEXT
QSTGS:SFW3.TEXT        changed to OLDSFW3.TEXT
```

Figure 4-11. Change Command Using Subgroup-Specifying String.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit
```

{The example below shows the use of the equal sign alone to change all the file names in a directory. The letter "A" is added before each file name.}

```
C
Change what file? LCALGS:=,a= <ret>
LCALGS:RGDEMO.RPGL   changed to ARGDEMO.RPGL
LCALGS:LC.CODE       changed to ALC.CODE
LCALGS:LCMASK        changed to ALCMASK
LCALGS:LCDUMP.CODE   changed to ALCDUMP.CODE
LCALGS:LLC.CODE      changed to ALLC.CODE
LCALGS:TEST.TEXT     changed to ATEST.TEXT
LCALGS:TEST2.TEXT    changed to ATEST2.TEXT
```

Figure 4-12. Change Command Using Equal Sign.

If the response to the "Change what file?" prompt in Figure 4-11 is changed to "LCALGS:a=,a <ret>", the "A" at the beginning of all the file names is removed.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

    {The command and response below changes the volume name "LCALGS:" to
    "QSTGS:".}

C
Change what file? LCALGS:,QSTGS: <ret>
LCALGS:              changed to QSTGS:
```

Figure 4-13. Change Command -- Changing the Volume Name.

4.2.8 T(rans (Transfer Command)

The Transfer Command copies the specified file(s) or volume to the given destination, leaving the source file or volume intact. The Transfer command is initiated by typing a T from the Filer main command line. The following prompt appears:

```
| Transfer what file?                                                    |
```

The source and destination for the copy must be given. These file specifications are required and must be separated by a comma or a <ret>. These specifications may be entered in response to the first prompt on that line separated by a comma. Alternately, the source specification only may be entered on that line; in that case, a second prompt appears as shown below.

```
-----------------------------------------------------------------------
| To where?                                                           |
-----------------------------------------------------------------------
```

The destination specification is required in response to the second prompt.

The size specification is recognized and is used to allocate space for the destination file. (See the Make command in this chapter.)

Transferring Files Across Volumes

An individual file or group of files can be transferred (or copied) from one volume to another, leaving the original file intact. Wild card specifications are valid in the file specifications. The following points describe the use of wild cards with the Transfer command.

- The $ can be used to transfer a file to another volume without changing the file name. The destination file name is replaced by the $ although the destination volume must still be given.

|NOTE|

The destination file name should not be completely omitted; the $ should appear with the volume name. If the file name is omitted, the directory of the volume may be destroyed. If the file name is omitted and no $ is given, the Filer prompts as below.

```
-----------------------------------------------------------------------
| Possibly destroy directory of <destination vol>?                    |
-----------------------------------------------------------------------
```

A "Y" answer to this prompt causes the directory of the destination volume to be destroyed. A "N" response allows the command to be reexecuted with the volume name plus a $.

- If the source file specification includes a wild card character and the destination is a block-structured device, the destination file specification must also contain the wild card character or must contain a $.

- Subset-specifying strings in the source specification are replaced with analogous strings (replacement strings) in the destination specification.

- Any of the subset-specifying strings may be empty. The equal sign (=) used alone specifies every file on the volume. This wild card character used as the destination specification causes the subset-specifying strings in the source specification to be replaced with nothing.

- The ? may be used in place of the equal sign to cause the Filer to prompt the user for confirmation of the transfer.

Transferring a File on the Same Disk

Files may also be transferred or copied from a volume to the same volume. To do so, the same volume name is specified for the source and destination. This capability is especially useful to relocate a file on the disk.

On same-disk transfer, specifying the number of blocks for the copied file causes the Filer to copy the file into the first available area that is at least as large as the specified size. Otherwise, the Filer copies the file into the largest unused area.

On a same-disk transfer, if the same file name is specified for both source and destination, the Filer rewrites the file to the size-specified area and removes the older copy of the file. (Two files with the same name cannot exist on the disk.) Thus, this type of transfer relocates a file with the original file name on the same disk and removes the old file.

|NOTE|

> Wild card characters should not be used in file specifications for any transfer on the same disk. The results are unpredictable.

The following prompts appear when the source and destination file names are given with the device number used as the volume specification.

```
--------------------------------------------------------------------
| Put destination disk in #5                                        |
| Type &lt;space&gt; to continue                                          |
--------------------------------------------------------------------
```

To effect the transfer on the same disk, a <space> is entered.

Transferring One Volume to Another

One complete volume is copied to another by specifying only the source and destination volume names or device numbers. Transfers from one Winchester volume to another Winchester volume result in a prompt that asks for a new name for the destination volume. Transferring from one block-structured volume to another causes the destination volume to be an exact copy of the source volume, including the directory. The following prompt appears to verify that an exact copy including directory is desired.

```
--------------------------------------------------------------------
| Possibly destroy directory of <destination volume>?               |
--------------------------------------------------------------------
```

If a Y or y is entered, the volume-to-volume transfer is completed. If an N is entered, the action is aborted, and the Filer main command line reappears.

The Y response is often used to create a backup copy of a source diskette. The name of the destination volume can be changed to show that it is a backup copy, if desired.

|NOTE|

> The name of the destination disk should be changed immediately, or the diskette removed, because two volumes on line with the same name cause unpredictable results.

Prior to the H2 release of the III.Ø Operating System, a volume-to-volume transfer did not transfer the bootstrap. However, with the H2 release, the Transfer command copies track Ø, where the bootstrap resides, to the new volume. This copy of track Ø occurs only on floppy disk transfers; transfers to or from the Winchester disk do not copy track Ø.

Transfers With Non-Block-Structured Volumes

The Transfer command can be used to copy files to volumes that are not block-structured (for example, CONSOLE:, PRINTER:, or REMOTE:) by specifying the appropriate volume name or device number. The file name is then ignored. The destination volume must be on line.

Transfers from non-block-structured devices are possible; however, the source must be an input device. In this case, the source file specification is unnecessary and is ignored if present.

Examples

Figures 4-14, 4-15, 4-16, 4-17, and 4-18 show examples of the Transfer command. Comments are enclosed in braces ({}), and user input is shaded.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

  {The following command and responses illustrate a transfer of all files
  using the equal sign and $.}
T
Transfer what file? LCALGS:=,LCALGS1:$ <ret>
LCALGS:RGDEMO.RPGL         transferred to LCALGS1:RGDEMO.RPGL
LCALGS:LC.CODE             transferred to LCALGS1.CODE
LCSLGS:LCMASK              transferred to LCALGS1:MASK
LCALGS:LCDUMP.CODE         transferred to LCALGS1:DUMP.CODE
LCALGS:LLC.CODE            transferred to LCALGS1:LLC.CODE
LCALGS:TEST.TEXT           transferred to LCALGS1:TEST.TEXT
LCALGS:TEST2.TEXT          transferred to LCALGS1:TEST2.TEXT
```

Figure 4-14. Transfer Command Using Equal Sign and $.

Figure 4-14 presents an example in which all the files on one volume are transferred to another volume using an equal sign to specify all files and a $ to specify that the files are copied with the same name as the original file.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

   {The following command and responses illustrate the use of a subset-
   specifying string to transfer a group of files to another volume.}
T
Transfer what file? LGCALGS:LC=,LOGIN:LL= <ret>
LGCALGS:LC.CODE              transferred to LOGIN:LL.CODE
LGCALGS:LCMASK               transferred to LOGIN:LLMASK
LGCALGS:LCDUMP.CODE          transferred to LOGIN:LLDUMP.CODE
```

Figure 4-15. Transfer Command Using Subgroup-Specifying Strings.

One, two or all three fields of the date entry may be changed. For example, entering 29 changes the day; entering -Jun changes only the month; and entering --83 changes only the year. The hyphens hold the place of the fields that are not changed. Also, entering 29-Jun changes the day and the month. (Any month name entered that is longer than three characters is truncated to three characters.)

If a <ret> is typed in response to the prompt, the date is not changed.

Figure 4-19 gives an example of changing the day and month through the Date command. Comments are enclosed in braces ({}); user input is shaded.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

   {The following command and responses illustrate changing the system date.}


D
Date set:<1..31>-<JAN..DEC>-<ØØ..99> or <CR>
Today is 5-May-82
New Date?  6-jun <ret>
New date is 6-Jun-82
```

Figure 4-19. Example of the Date Command.

In the Date command example, the month is entered with the first letter in lower case. When the Filer displays the new date the first letter of the month is an upper case character.

4.2.1Ø Q(uit (Quit Command)

The Quit command exits the Filer portion of the III.Ø Operating System; the main (outer level) command line is displayed on the top of the screen. This command is executed by typing Q from the Filer main command line.

To verify that the file was indeed moved, the E(xt-dir (Extended Directory) command can be used to see the size and location of the file before and after the transfer.

```
-------------------------------------------------------------------------
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit


   {The following command and responses illustrate a volume-to-volume transfer
   of files.}

T
Transfer what file? H3OS:,XXX: <ret>
Possibly destroy directory of XXX? Y

H3OS: transferred to XXX:
```

Figure 4-18. Transfer Command -- Volume-to-Volume Transfer.

The example in Figure 4-18 illustrates a volume-to-volume transfer of files. The new diskette is an exact copy of the source diskette. If an L (for List Directory) is entered for the new volume, the name of the new volume is H3OS, instead of XXX. The new diskette should be removed immediately or the volume name changed (see Change command) so that two volumes with the same name are not on line.

4.2.9 D(ate (Date Command)

The Date command sets (or changes) the date used by the III.Ø Operating System to show when a file is saved. The Date command is initiated by typing a D from the Filer main command line. The following prompt appears for the date change.

```
-------------------------------------------------------------------------
| Date set:<1..31>-<JAN..DEC>-<ØØ..99> OR <CR>                          |
| Today is 5-May-82                                                     |
| New date?                                                             |
-------------------------------------------------------------------------
```

A new date may be entered in the format described on the first line above, followed by a carriage return (<ret> or <cr>). The new date is immediately displayed.

Either a hyphen (-) or back-slash (/) may be used as the delimiter between the date fields.

Figure 4-15 presents an example in which the subgroup of files that begins with "LC" is transferred to another volume and is prefixed with the characters "LL" in place of the original "LC".

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

   {The following command and responses illustrate a file transfer across
   volumes.}

T
Transfer what file? LCALGS:TEST2.TEXT,LOGIN:TEST2.TEXT <ret>
LCALGS:TEST2.TEXT          transferred to LOGIN:TEST2.TEXT
```

Figure 4-16. Transfer Command -- File Transfer Across Volumes.

Figure 4-16 presents an example of copying one file from one volume to another volume using the file specification.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

   {The following command and responses illustrate relocating a file on the
   disk by using the same file specification for both the source and destin-
   ation.}

T
Transfer what file? LCALGS:LCMASK[9],LCALGS:LCMASK <ret>
LCALGS:LCMASK exists...remove it? Y
LCALGS:LCMASK transferred to LCALGS:LCMASK
```

Figure 4-17. Transfer Command -- Same Disk Transfer.

Figure 4-17 presents an example of relocating a file on the same disk. The size specification appears in brackets after the source file name. The Filer writes the new file into an unused area of at least nine blocks. If a size specification is not given, the Filer writes the new file in the largest unused area. The old (original) file is removed. The prompt regarding the removal of the existing file only appears if a size specification is used.

4.2.11 B(ad-blks (Bad Blocks Command)

The Bad Blocks command scans the disk/diskette to determine if damaged or corrupted blocks exist on the disk and to identify the bad areas. This command, although not displayed on the Filer main command line, may be executed by typing a B from the Filer main or secondary command line. (The secondary command line is displayed in response to the entry of a ? from the Filer main command line.)

The following prompt begins the bad-blocks scan.

```
| Bad blocks scan of what vol?                                          |
```

Either the volume name or device number, followed by a <ret>, is entered in response to the prompt.

After the first question is answered, the following prompt appears:

```
| Entire disk (nnnn blocks)?                                            |
```

This question requires a Y or an N as a response. The number in parentheses is the total number of blocks on the volume, which depends on how the disk was initialized and recorded. A Y (for yes) response causes the scan to begin. An N (for no) entered results in the following prompts.

```
| First block?                                                          |
```

The beginning block number of the range of blocks to be scanned is entered, followed by a <ret>. The next prompt asks for the ending block number of the range.

```
| Block nnnn to?                                                        |
```

The beginning block number of the range is substituted for nnnn above. The ending number is entered, followed by a <ret>. The following message then appears, restating the range to be scanned. The scan begins.

```
-------------------------------------------------------------------------
| Block range: nnnn to nnnn                                             |
-------------------------------------------------------------------------
```

If bad blocks are found, the prompts in the subsequent paragraphs appear.

Once the prompts for the entire disk or a range are answered, the scan begins. If bad blocks are found, the message below appears.

```
-------------------------------------------------------------------------
| Block nnnn is bad                                                     |
-------------------------------------------------------------------------
```

In the message, the nnnn represents the actual block number. Every bad block is reported on the display. Once a group of bad blocks are discovered, the following question appears.

```
-------------------------------------------------------------------------
| Examine blocks nnnn - nnnn?                                           |
-------------------------------------------------------------------------
```

This question presents the range of bad blocks (nnnn-nnnn) and asks if the user wants the bad blocks examined. If a Y is entered and the bad blocks contain data, the Filer supplies further information about the damaged area. This information is the name of the file written on the bad area; also, the block range of the file is reported. Another prompt appears as shown below.

```
------------------------------------------------------------------------
| File(s) endangered:                                                  |
|   file name     nnnn     nnnn                                        |
| Try to fix them?                                                     |
------------------------------------------------------------------------
```

The Filer can try to recover the blocks, or, if a fix is not possible, can mark the blocks as "bad". The attempt to fix the bad blocks consists of reading, rewriting, and then rechecking the blocks. If, after the attempt to fix the bad blocks, the blocks are still bad, the next prompt asks if the Filer should mark the bad blocks.

```
----------------------------------------------------------------------
| Mark them (may remove files!)?                                     |
----------------------------------------------------------------------
```

Marking bad blocks that have data stored on them causes the file to be removed; therefore, an n should be entered unless the file is expendable. If an n is entered, the scan continues.

However, if bad blocks are reported on an area of disk that is unused, those blocks can be marked and are not used in any subsequent writes to disk. In that case, if a Y response is entered, a quick message flashes as the blocks are marked. By executing the E(xt-dir (Extended Directory command), the user sees the area marked as bad. For example, the directory entry might appear as below.

```
--------------------------------------------------------------------------
| LCALGS:                                                                |
| <unused>          274                      10                          |
| BAD.00284.BAD     27 6-May-82      284      572 Bad Disk               |
--------------------------------------------------------------------------
```

A message appears when the scan is completed, reporting the total number of bad blocks, as below.

```
--------------------------------------------------------------------------
| nnnn bad blocks found                                                  |
--------------------------------------------------------------------------
```

If the disk/diskette contains bad blocks, the safest move is to transfer the good files from its directory to a good disk. The disk that has bad blocks can then be reformatted using the FORMAT program, then rescanned to determine if the area if usable.

In the previous sequence of prompts and actions, the examine and fix actions may report that the bad blocks were possibly fixed. Although the Filer may consider a block good, the block may not be "good" for the user's purpose. That is, a text file should be closely examined to determine if garbled text appears in the file. A data file should be manipulated by any program that uses it to determine the validity of the file. A code file that had bad blocks fixed should simply be replaced.

Often the cause of bad blocks is that bad or corrupted data were written on the disk. Overwriting or reformatting may correct that type of error. Physical damage to, or problems with the recording surface, are unrecoverable errors.

Figure 4-20 shows an example of a bad blocks scan in which bad blocks are reported. Comments are enclosed in braces ({}); user input is shaded.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

B
Bad block scan of what vol? H30S: <ret>
Entire disk (988 blocks)? Y
                                        {On floppy drives, clicking noises result
nnnn                                     as blocks are scanned.  Block numbers flash
Block 742 is bad                         in place of nnnn.}
Block 743 is bad
Block 744 is bad
Block 745 is bad
Block 746 is bad
Block 747 is bad
Block 748 is bad
Block 749 is bad
Block 750 is bad

Examine blocks 742-750? Y

File(s) endangered:
  SYSTEM.PASCAL    655   765            {File name and beginning and ending
Try to fix them? Y                       blocks of file.}
  Block 742 is bad
  Block 743 is bad
  Block 744 is bad
  Block 745 is bad
  Block 746 is bad
  Block 747 is bad
  Block 748 is bad
  Block 749 is bad
  Block 750 is bad

Mark them (may remove files!)? N
Continue scan? Y
Continue bad block scan
12 bad blocks found
```

Figure 4-20. Bad Blocks Scan (Bad Blocks Found).

Figure 4-21 gives an example of a bad block scan in which no bad blocks are found.

```
F(iler: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

B
Bad block scan of what vol? SBBK: <ret>
Entire disk (1976 blocks)? Y
nnnn                              {On floppy drives, clicking noises result
                                   as blocks are scanned.  Block numbers
                                   flash in place of nnnn.}

Ø bad blocks found
```

Figure 4-21. Bad Blocks Scan (No Bad Blocks Found).

4.2.12 E(xt-dir (Extended Directory Command)

The Extended Directory command is an extension of the List Directory command. This command displays, or lists, detailed information about the specified directory of a disk/diskette volume. Although this command is not displayed on the Filer main command line, it is executed by typing an E from the Filer main or secondary command line. (The secondary command line is displayed by typing a ? from the Filer main command line.)

The following prompt appears in response to the command invocation:

```
| Dir listing of what vol?                                              |
```

The use of wild card characters is the same for the Extended Directory command as for the List Directory command. (See Section 4.2.5 List Directory command.)

The data shown through use of this command are (1) file name; (2) unused areas of disk; (3) block length for each file; (4) last modification date; (5) starting block address; (6) number of bytes in the last block in the file; and (7) file kind. The summary line at the end of the list is the same as the summary line of the List Directory command.

Figure 4-22 gives an example of the Extended Directory command. Comments are enclosed in braces ({}); user input is shaded.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

E
Dir listing of what vol? LCALGS:<ret>

LCALGS:
RGDEMO.RPGL          8  12-Apr-82    10    96  Datafile
LC.CODE             67  12-Apr-82    18   512  Codefile
< UNUSED  >          9               85
LCDUMP.CODE         18  12-Apr-82    94   512  Codefile
LLC.CODE            64  25-Mar-82   112   512  Codefile
< UNUSED  >          8              176
TEST2.TEXT           4  30-Apr-82   184   512  Textfile
TEST.TEXT            4  30-Apr-82   188   512  Textfile
< UNUSED  >          9              192
LCMASK               9  30-Apr-81   201   512  Datafile
< UNUSED >        1766              210
7/7 files<listed/in-dir>, 174 blocks used, 1792 unused, 1766 in largest area
```

Figure 4-22. Example of the Extended Directory Command.

In Figure 4-22, the volume name appears at the top of the first column. The first column gives the file name and unused areas. Column two is the block length of the file. Column three is the last modification date of the file. The number in column four is the starting address of the file (block number). The fifth column shows how many bytes exist in the last block of the file. The last column shows the file kind.

4.2.13 K(rnch (Crunch Command)

The Crunch command moves the files on the specified volume toward the beginning of the disk so that unused blocks are grouped at the end. This command is initiated by typing K from the Filer main or secondary command line. (The command appears on the Filer secondary command line, which is accessed by typing a ? from the Filer main command line.)

The prompt that is displayed in response to this command follows.

```
| Crunch what vol?                                                      |
```

After the volume name or device number is entered, the following prompt is displayed.

```
-------------------------------------------------------------------------------
| Are you sure you want to crunch <vol name or ID>?                           |
-------------------------------------------------------------------------------
```

The second prompt asks the user to verify continuation of the crunch operation. A Y response causes the Filer to begin moving files. An N response aborts the crunch operation; the Filer main command line is redisplayed.

The volume specified to crunch must be on line. As each file is moved, its name is reported on the console. If SYSTEM.PASCAL is moved during a crunch operation, the system must be rebooted.

The Crunch command is not allowed if the Copier Task (see COPY utility, 6.16) is active on a file on the volume to be crunched.

Figure 4-23 presents an example of the Crunch command. Comments are enclosed in braces ({}); user input is shaded.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

  {In this example, the Extended Directory of the volume is listed before the
  Crunch command is used.  The crunch operation is completed; then, another
  copy of the Extended Directory listing is given.  Thus, a "before" and "after"
  picture of the disk usage is shown.}

E
Dir listing of what vol?  LCALGS:ret>

LCALGS:
RGDEMO.RPGL             8  12-Apr-82   10     96  Datafile
LC.CODE                67  12-Apr-82   18    512  Codefile
<  UNUSED  >            9              85
LCDUMP.CODE            18  12-Apr-82   94    512  Codefile
LLC.CODE               64  25-Mar-82  112    512  Codefile
<  UNUSED  >            8             176
TEST2.TEXT              4  30-Apr-82  184    512  Textfile
TEST.TEXT               4  30-Apr-82  188    512  Textfile
LCMASK                  9  30-Apr-82  192    512  Datafile
<  UNUSED  >         1766             210
7/7 files<listed/in-dir>  174 blocks used, 1792 unused, 1766 in largest area

K
Crunch what vol?  LCALGS:<ret>
Are you sure you want to crunch LCALGS:?  Y
Moving LLCDUMP.CODE
Moving LLC.CODE
Moving TEST2.TEXT
Moving TEST.TEXT
Moving LCMASK
LCALGS: crunched

E
Dir listing of what vol?  LCALGS:<ret>

LCALGS:
RGDEMO.RPGL             8  12-Apr-82   10     96  Datafile
LC.CODE                67  12-Apr-82   18    512  Codefile
LCDUMP.CODE            18  12-Apr-82   85    512  Codefile
LLC.CODE               64  25-Mar-82  103    512  Codefile
TEST2.TEXT              4  30-Apr-82  167    512  Textfile
TEST.TEXT               4  30-Apr-82  171    512  Textfile
LCMASK                  9  30-Apr-81  175    512  Datafile
<  UNUSED  >         1792             184
7/7 files<listed/in-dir>  174 blocks used, 1792 unused, 1792 in largest area
```

Figure 4-23. Example of the Crunch Command.

4.2.14 M(ake (Make Command)

The Make command creates a directory entry with the specified file name. This command is initiated by typing M from the Filer main or secondary command line. (This command is displayed on the secondary command line, which is accessed by typing a ? from the Filer main command line.)

The following prompt appears requesting the specified file name and specification.

```
-------------------------------------------------------------------------
| Make what file?                                                       |
-------------------------------------------------------------------------
```

The file specification must be entered in response to the prompt. In this case, the optional file size specification can be useful in managing disk space effectively. If no size specification is given, the Filer creates the file using the largest unused area of disk.

The size specification, if used, follows the volume name (or device number) plus file name. That is, the number of blocks enclosed in brackets ([]) appears immediately to the right of the file name. Two default size specifications are explained below.

[0] This size specification is the same as omitting a size specification. The file is created in the largest unused area.

[*] This size specification results in the file being created in the second largest, or half the largest unused area, whichever is larger.

Because other files cannot use the area allocated to a file created by the Make command, the command can be used to create a directory entry in order to reserve that area of disk (for example, to save the space for future use).

Files with a file name that ends with .TEXT must occupy at least four blocks and must occupy an even number of blocks. If the Make command is used to create a text file with a size specification of less than four blocks, the following message appears:

```
-------------------------------------------------------------------------
| No room on vol                                                        |
-------------------------------------------------------------------------
```

If the Make command is used to create a .TEXT file with a size specification that is an odd number of blocks, the file is created with one less block than specified.

Figure 4-24 gives examples of the M(ake command to create file entries in the directory. In the figure, comments are enclosed in braces ({}); user input is shaded.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

  {Three examples of the Make command are shown below.  The Extended Directory
  listing for the volume is shown before and after the Make operation.}

E
Dir listing of what vol? LCALGS:<ret>

LCALGS:
RGDEMO.RPGL          8  12-Apr-82   10     96  Datafile
LC.CODE             67  12-Apr-82   18    512  Codefile
LCDUMP.CODE         18  12-Apr-82   85    512  Codefile
LLC.CODE            64  25-Mar-82  103    512  Codefile
TEST2.TEXT           4  30-Apr-82  167    512  Textfile
TEST.TEXT            4  30-Apr-82  171    512  Textfile
LCMASK               9  30-Apr-81  175    512  Datafile
<  UNUSED  >      1792             184
7/7 files<listed/in-dir>  174 blocks used, 1792 unused, 1792 in largest area

M
Make what file? LCALGS:TEST3.TEXT[28]<ret>
LCALGS:TEST3.TEXT  made
M
Make what file? LCALGS:TEST4.TEXT[5]<ret>
LCALGS:TEST4.TEXT  made
M
Make what file? LCALGS:TEST5.TEXT[*]<ret>
LCALGS:TEST5.TEXT  made

E
Dir listing of what vol? LCALGS:<ret>

LCALGS:
RGDEMO.RPGL          8  12-Apr-82   10     96  Datafile
LC.CODE             67  12-Apr-82   18    512  Codefile
LCDUMP.CODE         18  12-Apr-82   85    512  Codefile
LLC.CODE            64  25-Mar-82  103    512  Codefile
TEST2.TEXT           4  30-Apr-82  167    512  Textfile
TEST.TEXT            4  30-Apr-82  171    512  Textfile
LCMASK               9  30-Apr-81  175    512  Datafile
TEST3.TEXT          28   6-May-82  184    512  Textfile
TEST4.TEXT           4   6-May-82  212    512  Textfile
TEST5.TEXT         880   6-May-82  216    512  Textfile
<  UNUSED  >       880            1096
10/10 files<listed/in-dir>, 1086 blocks used, 880 unused, 880 in largest area
```

Figure 4-24. Three Examples of the Make Command.

The three examples of the Make command in Figure 4-22 illustrate several points. The first make operation creates a file that is allocated 28 blocks of space; the Extended Directory listing confirms that the specified size was allocated for the file TEST3.TEXT.

The second make operation specified the creation of a text file of five blocks. The file was actually allocated four blocks, one block less than the odd-numbered size specification, as shown on the directory listing.

The third make operation uses a default size specification that causes the Filer to allocate half the largest unused area to the file. The directory listing confirms that the file entry is made according to specification. The "before" listing shows a total, and the largest unused area, of 1792 blocks. The first two make operations used 32 of those blocks, leaving 1760 blocks in the largest unused area. The third make operation specifies a size of half that area -- or 880 blocks. The directory listing confirms that 880 blocks are allocated for the file TEST5.TEXT.

4.2.15 P(refix (Prefix Command)

The Prefix command changes the current default volume to the volume specified. The default (prefixed) volume is the volume on which the III.0 Operating System searches for any file referenced which does not have an explicit volume name or device number given. By default, the system volume is the prefixed volume unless changed by the Prefix command.

The Prefix command is executed by typing a P from the Filer main menu or secondary command line. (This command appears on the Filer secondary command line, which is accessed by typing a ? from the Filer main command line.) The prompt in response to a Prefix command is as follows:

```
Prefix titles by what vol?
```

The desired volume name or device number is entered in response to the prompt. If the current default prefix is not known, entering a colon (:) in response to the prompt causes the current prefixed volume to be identified.

After the volume name or number is entered, a message is displayed as follows:

```
--------------------------------------------------------------------------
| Prefix is <volume name/device number>                                  |
--------------------------------------------------------------------------
```

The volume specified to be the prefixed volume is not required to be on line.

If the volume prefix is changed to other than the system volume and the system is rebooted for some reason, the prefixed volume is still the default (the system volume) when the system comes up after rebooting.

Figure 4-25 gives an example of the Prefix command. Comments are enclosed in braces ({}); user input is shaded.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, T(rans, D(ate, Q(uit

P
Prefix titles by what vol?  :<ret>
Prefix is H3OS:

P
Prefix titles by what vol? LCALGS:<ret>
Prefix is LCALGS:

P
Prefix titles what what vol? :<ret>
Prefix is LCALGS:
```

{The above commands show first the current prefixed volume, which is the operating system volume H3OS. The next Prefix command changes the prefixed volume to LCALGS:. The third execution again checks the current prefixed volume, which is now LCALGS: instead of H3OS:.}

Figure 4-25. Examples of the Prefix Command.

4.2.16 V(ol (Volumes Command)

The Volumes command lists all volumes currently on line and, also, shows some reserved volumes that are not on line. The on-line volumes show the associated unit numbers. No prompt line is displayed; no file specification is allowed.

The Volumes command is specified by typing V from the Filer main or secondary command line. (This command is displayed on the Filer secondary command line, which is accessed by typing a ? from the Filer main command line.)

Figure 4-26 shows a typical display produced by the Volumes command. In the figure, the user input is shaded.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

V

   Volumes available -- '(number)' indicates unit offline
   ------------------------------------------------------
Serial I/O

  1 CONSOLE:          2 SYSTERM:  |    7 RCONS1:       8 REMOTE:   |
(15) RCONS2:       (16) RTERM2:   |  (17) RCONS3:    (18) RTERM3:   |

Parallel I/O

(6) PRINTER:

Blocked I/O -- ':'  Prefix is LCALGS: '*' System unit is H3Ø5:

 4 H3Ø5:   5 LCALGS:
```

Figure 4-26. Typical Volumes Command Display.

The display in Figure 4-26 lists information about on- and off-line volumes. (See Section 2.2 also.) The on- and off-line volumes with associated unit/ devices numbers are listed.

The serial I/O non-block-structured volumes are (1) the system console (unit #1: or CONSOLE:); (2) the system terminal (unit #2: or SYSTERM:), which is the computer communication line back to the system console; (3) a serial printer (unit #8: or REMOTE:); (4) a remote console (unit #7: or RCONS1:), which is the computer communications line to the serial; and (5) two sets of off-line remote terminals and remote consoles (unit #s 15:/16:, and 17:/18: — RCONS2:/RTERMS2: and RCONS3:/RTERM3:), which are remote devices (each with a system communication line from the computer) which could be connected to this 1600 modular system. Additional remote devices could be added such that the volume pairs 19:/20: through 25:/26: would be used for RCONS4:/RTERM4 through RCONS7:/RTERM7.

For parallel I/O, the printer (unit #6: or PRINTER:) is off line. Volume 27 could be an additional parallel printer (PRINTR1:).

The block-structured I/O volumes are unit #4: (H305:) and unit #5: (LCALGS:). The prefixed volme is currently LCALGS:, and the system unit is H305:. Additional block-structured volumes (unit #s 9 through 14) are allowed.

4.2.17 X(amine (Examine Command)

The Examine command attempts physical recovery of suspected bad blocks that have been detected by use of the Bad Blocks command. (See Section 4.2.11.) The Examine command is initiated by typing X from the Filer main or secondary command line. (This command is displayed on the Filer secondary command line, which is accessed by typing a ? from the Filer main command line.)

After an X is entered, the following prompt appears:

```
Examine blocks on what vol?
```

The volume name (or device number) entered must be on line. After the volume name or device number is entered, the following prompt appears:

```
First block?
```

After the first block of the range of blocks is entered, the following prompt appears:

```
Block nnnn to?
```

After the ending block number of the range is entered, the following message appears and the Examine action begins.

```
| Block range: nnnn to nnnn                                            |
```

The Examine action reads the bad block to determine if it is bad. If the Examine message reports that the block(s) is bad and the block has been written to, the message below appears:

```
| File(s) endangered:                                                  |
|  <file name>                                                         |
| Try to fix them?                                                     |
```

A Y answer causes the Filer to examine the blocks and return either of the following messages:

```
| Block nnnn may be OK                                                 |
| Block nnnn is bad                                                    |
```

In the first case, the block may have physically been fixed. In the second case, an option to mark the block as bad is given. The prompt regarding marking the block is as below.

```
| Mark them?                                                           |
```

If the bad blocks occur on an unused area of disk, marking the blocks as bad prevents that area from being used by other files or from being used by the Crunch command.

A N answer to the "Try to fix them?" prompt causes the Filer main command line to be displayed.

Although the block may be reported as possibly OK, it can be physically OK but can contain garbage. Fixing a block consists of reading, writing, and rereading the block. If the reads are the same, the message "May be OK" is displayed. If the reads are different the block is declared bad.

Figure 4-27 gives examples of the Examine command to examine bad blocks on a disk that contains bad blocks. Comments are enclosed in braces; user input is shaded.

|NOTE|

Western Digital has enhanced the Bad Blocks command to include the Examine function as an automatic response to detecting bad blocks. However, the ability to perform the Examine function separately is still available.

```
Filer: G(et, S(ave, W(hat,N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

  {A disk containing no files is known to have three areas of bad disk.
  One area has been marked as bad.  The Examine command is used below to
  determine if the other areas are bad and, if so, to mark them.  An
  Extended Directory listing is given to show the areas marked as bad.}

X
Examine blocks on what vol?  #5:<ret>
First block?  336
Block 336 to?  336
Block range: 336 to 336
Block 336 is bad
Block 336 is still bad
Mark them?  Y
BAD.ØØ336.BAD marked

X
Examine blocks on what vol?  #5:<ret>
First block?  388
Block 388 to?  414
Block range: 388-414
Block 388 is bad
Block 389 may be OK
       .
       .
       .
Block 413 may be OK
Block 414 is bad
Blocks 388 thru 414 are still bad
Mark them?  Y
BAD.ØØ388.BAD marked

E
Dir listing of what vol?  #5:<ret>

LCALGS:
<  UNUSED  >           274                    1Ø
BAD.ØØ284.BAD           27  6-May-82         284     512  Bad disk
<  UNUSED  >            25                   311
BAD.ØØ336.BAD            1  6-May-82         336     512  Bad disk
<  UNUSED  >            51                   337
BAD.ØØ388.BAD           27  6-May-82         388     512  Bad disk
<  UNUSED  >          1561                   415
3/3 files<listed/in-dir>,  55 blocks used,  1911 unused,  1561 in largest area
```

Figure 4-27. Examples of the Examine Command.

4.2.18 Z(ero (Zero Command)

The Zero command is used to initialize the directory on a specified volume with a new volume name and with all blocks on the disk unused. The Zero command is initiated by typing Z from the Filer main or secondary command line. (This command is displayed on the secondary command line, which is accessed by typing a ? from the Filer main command line.)

The following prompt appears in response to the Z entered.

```
---------------------------------------------------------------------------
| Zero dir of what vol?                                                   |
---------------------------------------------------------------------------
```

The current name (or device number) of the volume to be zeroed is entered; the volume must be on line. If the disk has not previously been zeroed, the following prompt appears:

```
---------------------------------------------------------------------------
| Duplicate dir?                                                          |
---------------------------------------------------------------------------
```

If a Y response is entered, a duplicate directory is maintained. The primary directory resides in blocks 2-5; the duplicate directory resides in blocks 6-9. If the primary directory is destroyed, the disk can be restored from the duplicate directory using the utility COPYDUPDIR.

The next prompt appears as below:

```
---------------------------------------------------------------------------
| # of blocks nnnn (max)?                                                 |
---------------------------------------------------------------------------
```

This prompt shows the maximum number of blocks that can be entered on the disk depending on the disk type. Table 4-2 lists the various types of diskettes and the appropriate number of blocks for each.

Table 4-2. Block Quantities on Disk.

Disk Type	Number of Blocks
Single-density, single-sided, soft-sectored, 8" floppy	494
Single-density, dual-sided, soft-sectored, 8" floppy	988
Double-density, single-sided, soft-sectored, 8" floppy	988
Double-density, dual-sided, soft-sectored, 8" floppy	1976

The user must answer with a Y or an N; if an N is entered, a prompt appears asking for the number of blocks. The Filer next prompts for the new volume name as follows:

```
New vol name?
```

After the volume name is entered, the following question appears for verification that the data are correct.

```
<volume name> nnnn blocks correct?
```

If a Y is entered, the diskette is zeroed, and the following message appears.

```
<volume name> zeroed
```

If an N response is given, the zero action is aborted, and the Filer main command line appears.

If the specified disk has been previously zeroed, the following changes in the prompt sequence occur. The following prompt appears before the duplicate directory prompt:

```
Destroy <volume name>?
```

This prompt asks for verification that the directory of the volume is to be zeroed.

Also, the prompt regarding the number of blocks is different, as shown below:

```
| Keep present # of blocks <nnnn>?                                    |
```

Figures 4-28 and 4-29 show examples of the Zero command and the appropriate responses. Comments are enclosed in braces ({}); user input is shaded.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

   {The following command and responses are appropriate for zeroing a blank
   diskette.  The diskette is formatted (Section 6.12) before the diskette
   is zeroed.}

Z
Zero dir of what vol?  #5:<ret>
Duplicate dir?  Y
# of blocks 1976 (max)?  Y
New vol name?  SFW
SFW: (1976 blocks) correct?  Y
SFW zeroed.
```

Figure 4-28. Zero Command (Blank Diskette).

The above diskette is a double-sided, double-density diskette. The number of blocks must be entered in response to the prompt although the appropriate number for the type of diskette appears as a reminder.

```
Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit

   {The following commands and responses are a typical sequence for zeroing
   a single-sided, double-density diskette that has been previously zeroed.}

Z
Zero directory of what vol?  LC:<ret>
Destroy LC?  Y
Keep present # of blocks <988>?  Y
New vol name?  BKUP:<ret>
BKUP: (988 blocks) correct?  Y
BKUP: zeroed
```

Figure 4-29. Zero Command (Disk Previously Zeroed).

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

5. PASCAL COMPILER

The III.Ø Operating System Pascal Compiler converts source programs (English-like statements) into machine-executable P-code (or instructions).

This Compiler supports UCSD Pascal(TM) developed at the University of California, San Diego. UCSD Pascal includes the Wirth nucleus plus some additional features that expand its capabilities. Western Digital has also added features to the Compiler. Some of the extensions to the III.Ø Operating System Compiler are listed below:

- Long integers (up to 36 digits).
- Strings.
- Random file access.
- Automatic loading of program segments from disk storage.
- Separate compilation and linking of Pascal modules.
- I/O and interrupt programming.
- Program synchronization through SIGNAL and WAIT ON SEMAPHORE.
- Multitasking through START of Pascal processes.

The Compiler suports the 128-segment capability on static (system segment vector) as well as vectored (user segment vector) code files. Programs compiled on the H2-release Compiler may not be run on software releases prior to H2. However, programs compiled on releases prior to the H2 release will run on the H2 or later version of the operating system.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

5.1 COMPILER INVOCATION

The Pascal Compiler is a one-pass recursive descent compiler invoked from the III.Ø Operating System outer level command line by typing a C (for C(ompile) or an R (for R(un).

The Compile command is used to explicitly compile a source file. If a work file exists, the Compiler automatically compiles the work file. If no work file exists, the Compiler prompts for the name of the file to be compiled; the prompt is shown below:

```
------------------------------------------------------------------------
|Compile what text? (<ret> to exit)                                    |
------------------------------------------------------------------------
```

Likewise, the Run command automatically compiles the work file. The Run command is used most often with the SYSTEM.WRK.TEXT file. The work file is created which contains the source code; then the Run command is used to compile and execute the source program. An attendent SYSTEM.WRK.CODE file results from the compilation if no syntax errors are detected.

If a syntax error is detected, the Compiler indicates the area in the source text where the error occurred. (Section 5.4.2). The user has three choices regarding the syntax error: (1) continue the compilation; (2) terminate the compilation, or (3) return to the Editor to change the source in question.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

5.2 SYNTAX OF COMPILER OPTIONS

Certain compiler options are available that instruct the Compiler to generate code according to a given specification (see section 5.3). The Compiler options are written as comments in the source text; these options are preceded by a $. The syntax of these options can be one of the two following formats:

(*$<option sequence><any comment>*)

{$<option sequence><any comment>}

In these two formats the (**) and {} enclose comments.

In the <option sequence> portion of the syntax, the options are listed separated by commas. Each option is shown by a capital letter followed by either a plus (+) sign or minus (-) sign. The plus activates the option; the minus negates the option. The letter designating the first option must immediately follow the $.

If default options are to be used, they are not included in the <option sequence>. Three of the compiler options described in Section 5.3 may be followed by file names rather than a plus or minus. These options are (1) the Included (I) option, which includes another source file in the compilation; (2) the Listing (L) option, which lists the source program to a nondefault destination (for example, to another file); and (3) the User Program (U) option, which determines whether or not the compilation is of a user or system program.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

5.3 COMPILER OPTIONS

As explained previously, certain compiler options are available to allow the user to specify how the compilation of the source program is to proceed. This section lists and describes those compiler options. All compiler options may be entered in upper or lower case.

5.3.1 G Option (Accept/Reject GOTO Statements)

The G (GOTO) option determines whether or not the Compiler accepts a GOTO statement. The effects of the plus and minus sign when used with "G" are shown below.

G- Generates a syntax error on encountering a GOTO statement.

G+ Allows the use of the GOTO statement in the source text.

| NOTE |

> In a Pascal program where a GOTO statement is used, other statements may often be substituted. For example, the FOR, WHILE, or REPEAT statements may be used in most cases.

The default value for this option is G-.

5.3.2 I Option (Include Another Source File)

The I (Include) option has two basic forms. The form in which the I is immediately followed by a plus or minus controls I/O checking code that is emitted by the Compiler. The other form in which the I is immediately followed by a file name causes the Compiler to include a different source file into the compilation at the point.

I/O Checking

The default value for the I/O checking form of this option is I+.

The effects of the plus and minus signs used with this option are given on the following page:

I+ Generates code after each I/O statement to determine whether or not the I/O completed sucessfully. If the I/O does not complete, the program terminates with a run-time error.

I- Does not generate I/O checking code. Therefore, if an I/O does not successfully complete, the program does NOT terminate with a run-time error.

The I-option is used frequently with system level programs which explicitly check the IORESULT function after each I/O operation. The I- option program can detect and report I/O errors without terminating abnormally. However, if IORESULT is not checked with the I- option, I/O errors may be undetected, and program bugs may increase.

Include Another Source File

The syntax of the Include option that instructs the Compiler to include another source file is given below. Either form may be used.

```
(*$I<file name>*)

{$I<file name>}
```

The comment must be closed at the end of the file name; no other options can follow. If the file name begins with a plus or minus sign, a blank must be inserted between the I and the file name.

If the first attempt to open the included file fails, the Compiler appends ".TEXT" to the name and tries again. If the second attempt fails or if an I/O error occurs while reading the included file, the Compiler responds with a fatal syntax error.

The included file may be inserted at any point in the original program on the condition that the rules governing the normal order of Pascal declarations is not violated within a given file. The Compiler accepts included files that contain the declarations CONST, TYPE, and VAR, even when the file has partially or fully completed its declarations.

If the included file contains PROCEDURE or FUNCTION declarations, the "I" comment must appear after all CONST, TYPE, and VAR declarations in the original file. If the included file contains CONST, TYPE, or VAR declarations, the "I" comment must appear before the first PROCEDURE or FUNCTION declaration of the original program.

The Compiler cannot track nested Include comments. Therefore, if the included file contains another Include comment, a fatal syntax error is generated.

The Include option is useful for breaking up large programs in smaller, more easily managed parts.

SYSTEM.SWAPDISK

SYSTEM.SWAPDISK is an empty file that is used by the Compiler to swap out information in its table when compiling very large programs. This file is used only when compiling programs that use the I option and contain many variables.

5.3.3 L Option (Source Program Listing)

The L (listing) option directs the Compiler either to generate a listing of the source program to a given file or not to generate a listing. The default is L-.

The effects of the plus and minus sign used with this option are described below:

- L- Does not generate a compiler listing.
- L+ Generates a compiler listing and writes the listing in a disk file "SYSTEM.LST.TEXT".

The user may override the default destination by specifying a file name following the L. To specify a file name with an option comment, refer to the description in Section 5.3.2, the I Option.

| NOTE |

> The file that contains the program listing may be edited the same as any other file if the file name contains the suffix ".TEXT". Otherwise, the file is considered to be a data file rather than a text file.

The contents of a source program listing are described and illustrated in Section 5.4.3.

5.3.4 P Option (Paging a Listing)

The P option causes a source program listing to be paginated. That is, a page feed is generated every time a P+ occurs in the source text. The effect of the plus and minus used with this option are described below:

- P- Suppresses paging.
- P+ Generates a page in a source program listing.

The default value for this option is P-.

5.3.5 Q Option (Quiet Compiler)

The Q option ("quiet" compiler) determines whether or not the Compiler generates procedure names and line numbers detailing the progress of compilation at the system console.

The effects of the plus and minus signs used with this option are described below:

- Q+ Suppresses compilation progress information to the CONSOLE device as compilation progresses.
- Q- Sends procedure names and line numbers to the CONSOLE device as compilation progresses.

The default value for this option is set to the current value of the SLOWTERM attribute of the system communication record, SYSCOM^.MISINFO. SLOWTERM. (Refer to Section 6.1 for setting this attribute.)

5.3.6 R Option (Range Checking)

The R option (range checking) determines whether or not additional code is generated to check array subscripts and assignments to variables of subrange types. The default is R+.

The effects of the plus and minus signs used with this option are described below:

- R+ Enables range checking.
- R- Disables range checking.

| NOTE |

> Programs compiled with the R- option selected usually run slightly faster than those compiled with the R+ option selected. However, if an invalid index or an invalid assignment is made, the program does NOT terminate with a run-time error.

5.3.7 S Option (Swapping Mode)

The S option determines whether or not the Compiler operates in swapping mode. In swapping mode, only one of two main parts (declarations or statements) is in main memory at one time. Swapping out one main part frees 2500 additional words of memory for symbol table storage.

A consideration, however, is that in swapping mode compilation is slower. Generally the compilation time is two-to-three times slower in swapping mode than in nonswapping mode. This option must be set before the Compiler encounters any Pascal syntax.

The effects of the plus and minus signs used with this option are described below. The default value for this option is S-.

- S+ Puts the Compiler in swapping mode.
- S- Puts the Compiler in nonswapping mode.

5.3.8 U Option (User Program)

The U option determines whether or not the compilation is a user or system program compilation. The default value for the option is U+. The effects of the plus and minus signs used with this option are described below:

- U+ Compiles the program at a user program lexical level.
- U- Compiles the program at the system lexical level. This form of the U option sets the R-, G+ and I- options.

| NOTE |

> Selecting U- generates programs that may not behave as expected. The U- option is not recommended for nonsystem work unless the method of operation is known.

Prior to the H3 release of the operating system, programs compiled with $U- option were restricted to 16 segments (Ø to 15). With the H3 release, these programs may contain up to 128 segments (Ø to 127).

| NOTE |

> Because $U- programs use the system segment vector at run time, they must coexist nondestructively with the operating system segments. Thus, to avoid replacing system segments by the $U- program segments, all $U- programs should declare dummy segments for any segments in use by the operating system. The only segments guaranteed NOT to be operating system segments are segments 1, and 8..15. Therefore, any $U- program that includes segments other than these may not be capable of coexisting nondestructively with the operating system.

Figure 5-1 shows an example of a program that uses the $U- option. Because the U- option allows access to operating system globals, use of the option can be dangerous.

```
{$U-}
{This program demonstrates $U-.  This compiler option allows a programmer to
access operating system globals.  Be careful about altering operating system
globals as this can have a deleterious effect.  This option also allows
dynamic allocation of files in the heap.}

program pascalsystemexample;

type
  phyle = file;
  inforec = record
             worksym,workcode: ^phyle;
             errsym,errblk,errnum: integer;
             slowterm,stupid: boolean;
             altmode: char;
           end;
var filler: array[Ø..6] of integer; {space holder for unused OS globals}
    userinfo: inforec;

segment procedure theprogram; {This segment procedure is the actual user
program.  The program's global variables should be declared here.}

type filep = ^phyle;

var cp: filep;
    arr: packed array[Ø..1Ø23] of char;
    c: char;
    fil: file;

{Declare 8 segment procedures with no code to make subsequent segment pro-
 cedures fall in the user segments. This is necessary as the operating system
 uses segments Ø and 2-7, while a user program has segments 1 and 8-15.  These
 'forward' declarations are only needed if the program contains other segment
 procedures. Note that $U- allows forward procedures to remain unresolved,
 because they are needed only as space holders.}

 segment procedure num2; forward;
 segment procedure num3; forward;
 segment procedure num4; forward;
 segment procedure num5; forward;
 segment procedure num6; forward;
 segment procedure num7; forward;

 {The program's segment procedures, if any go here,}
 segment procedure firstuserses;
 var i:integer;
 begin
  writeln (' in segment 8 ');
  i := i + 1;
 end;
```

Figure 5-1. $U- Option Example (Continued on Next Page).

```
begin

{This code is invoked when this program is executed.}
{In other words, this code will be the outerblock of the program.}
{For example, get the altmode character defined by SETUP }
 c := userinfo.altmode;

  {Dynamically allocate a file.}
   new(cp);

  {The following statement moves the initialized FIB fil (done automatically
   by the system) into the new FIB for cp.}
   moveleft(fil,cp^,sizeof(fil));

   reset (cp^,'dum.text');
   if blockread(cp^,arr,2) <> 2 then writeln ('read error');

   {Call the first user segment procedure.}
   firstusers;
end;
begin end.  {This code will never be executed.}
```

Figure 5-1. $U- Option Example (Continuation).

The U option is also used to name a library file. The file named becomes the file in which subsequent USES UNITs are sought. The default file for the library is *SYSTEM.LIBRARY.

Figure 5-2 gives an example of a USES clause with the U option.

```
USES UNITA, UNITD,    {Found in *SYSTEM.LIBRARY}
  {$U NEW.CODE}
     UNITB,
  {$U OLD.CODE}
     UNITC,UNITE;
```

Figure 5-2. Example of USES Clause with U Option.

The example code causes the Compiler to read units UNITA and UNITD from the file *SYSTEM.LIBRARY, unit UNITB from the file NEW.CODE, and units UNITC and UNITD from the file OLD.CODE. The ".CODE" suffix must be included as part of the file name for the alternate library.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

5.4 COMPILER OUTPUT

The compilation process can produce several types of output: (1) compilation progress displayed on console; (2) syntax error messages; and (3) a source program listing.

5.4.1 Compilation Status Information

During the course of compilation, the Compiler displays messages on the system console detailing the progress of the compilation. The information can be supressed (described in this section) so that no information is displayed.

Information Displayed

The following compilation progress information is displayed on the console unless the display is explicitly supressed.

- Name of the file being compiled.
- The message: Compiling....
- Name and version of the Compiler.
- Line number of the first line compiled, enclosed in angle brackets (<>).
- A dot (.) for each line compiled.
- At the start of compilation of the first procedure:
  - -- Line number, enclosed in angle brackets.
  - -- Identifier of the procedure.
  - -- Number of 16-bit words currently available for symbol table storage, enclosed in square brackets ([]).
  - -- A dot as each source line is compiled. Whenever 50 lines have been compiled, the Compiler generates the current line number.
- For each subsequent procedure encountered, the same items listed above.
- When all source lines have been compiled:
  - -- The total number of lines compiled.
  - -- A message indicating the smallest available space.

Figure 5-3 shows a typical display of compilation status information.

```
SEND.TEXT

Compiling...



       PASCAL Compiler [3.Ø]
       <   Ø>....................... ..................
       LAINIT [471Ø]
       <  43>.........
       GETFILE [4692]
       <  52>..................
       WRITEIT [4674]
       <  71>.............
       NEWLINE [4634]
       <  84>............ ................... ........... ........
       < 134>.................................................
       < 184>........
       COPYIT  [4616]
       < 192>............ ...
       SEND [4627]
       < 2Ø5>.....


      211 lines
      Smallest available space = 4616 words
```

Figure 5-3. Typical Compilation Status Information.

In Figure 5-3, a 211-line program is read from the file SEND.TEXT and compiled. The program contains six procedures.

Supressing Compilation Status Information

If the source program includes the Q+ (quiet compile) option, the display of compilation status information is surpressed. (See Section 5.3.5.)

5.4.2 Syntax Error Messages

If a syntax error is detected in the source program, the Compiler generates an error message. A list of the error messages is given in Appendix B.2.

A syntax error message consists of the segment of source text containing the error, with the symbol at which the error was detected indicated by the marker:

<<<<

The error number follows the text. (See Appendix B.2)

Although error messages are usually displayed on the screen, if the source program contains the Q+ (quiet compile) and L+ (list source program) options, syntax error messages are written to the file SYSTEM.LST.TEXT (which may be edited like any text file) and compilation continues.

If these options are not selected and a syntax error is encountered, the Compiler generates the error message to the console device. The Compiler then prompts the user to indicate how to proceed, for example:

```
Line N, error nnn:<sp>(continue),<esc>(terminate), E(dit
```

The response to the prompt is to enter a space, an escape, or an E (for edit). These three options cause the following actions:

<space> Continue with the compilation.

<esc> Terminate the compilation.

E Access the editor.

If the Editor is accessed, the cursor is positioned at the error location and the error number is displayed at the top of the screen. All Editor facilities are available to correct the error. Once the error is corrected, the compilation can be repeated.

Figures 5-4 and 5-5 show examples of syntax errors. The example in Figure 5-4 lists the error message to the screen; the example in Figure 5-5 writes the error message to the SYSTEM.LST.TEXT file. Comments within the figures are enclosed in brackets.

{The following simple Pascal program is entered as the system work file SYSTEM.WRK.TEXT). The period after the word "END" is omitted (a syntax error).}

```
PROGRAM SUM(INPUT,OUTPUT);
{*THIS IS A SIMPLE PASCAL PROGRAM.*)
var A,B,C,D,TOTAL:INTEGER;
BEGIN
  WRITELN('ENTER FOUR NUMBERS TO BE ADDED...');
  READ (A,B,C,D);
  TOTAL := A + B + C + D;
  WRITELN('TOTAL EQUALS', TOTAL)
END
```

{When the Run command is executed, the following display appears.}

```
Compiling.....

PASCAL Compiler [H3]
-->SYSTEM.WRK.TEXT
<   Ø>....
SUM        [3Ø267]
<   4>....
-- SYSTEM.WRK.TEXT
   <<<<
Line 9, error 4Ø1:<sp>(continue),<esc>(terminate),E(dit
```

Figure 5-4. Syntax Error Example - Message displayed.

In Figure 5-4, if an E is entered, the Editor brings up the work file with cursor positioned on line 9 and the error number displayed at the top of the screen. Error 4Ø1 is defined in Appendix B.2 as "Unexpected end of input".

{The following simple Pascal program is entered as the system work file SYSTEM.WRK.TEXT. The semicolon after "INTEGER" is omitted (a syntax error). The quiet compile (Q+) and list source program (L+) options are selected.}

```
 PROGRAM SUM(INPUT,OUTPUT);
{*THIS IS A SIMPLE PASCAL PROGRAM.*}
{$Q+,L+}
VAR A,B,C,D,TOTAL:INTEGER
BEGIN
  WRITELN('ENTER FOUR NUMBERS TO BE ADDED...');
  READ (A,B,C,D,);
  TOTAL := A + B + C + D;
  WRITELN('TOTAL EQUALS', TOTAL)
END.
```

{When the Run Command is executed, the following display appears.}

```
Compiling....



PASCAL Compiler [H3]
--> SYSTEM.WRK.TEXT
<   Ø>...
—   SYSTEM.WRK.TEXT
1Ø lines, 4 sec,15Ø lines/min
```

{Because the Q+ and L+ options are selected, the error message is written to the SYSTEM.LST.TEXT file, which is shown below.}

```
     3  128    1:D     1 {$Q+,L+}
     4  128    1:D     1 VAR A,B,C,D,TOTAL:INTEGER
     5  128    1:Ø     Ø BEGIN
>>>>>> Error # 14
     6  128    1:1     Ø   WRITELN('ENTER FOUR NUMBERS TO BE ADDED...');
     7  128    1:1    25   READ (A,B,C,D);
     8  128    1:1    69   TOTAL := A + B + C + D;
     9  128    1:1    78   WRITELN('TOTAL EQUALS', TOTAL)
    1Ø  128    1:Ø   11Ø END.
```

Figure 5-5. Syntax Error Example - Q+ and L+ Option Selected.

In Figure 5-5, the error message is written to the SYSTEM.LST.TEXT file. Error #14 is defined in Appendix B.2 as " ';' expected". The source program listing is described in Section 5.4.3.

5.4.3 Source Program Listing

If the source program includes the L or L <file> option, the Compiler generates a source program listing.

Preceding each source line in the program listing are five pieces of information:

- The line number.
- The segment procedure number.
- The procedure number.
- Either
  - -- A letter D to indicate that the line is part of the declarations, or
  - -- An integer (Ø through 9) to denote the lexical level of statement nesting within the code part.
- The number of bytes (for code) or words (for data) required by the declarations of the procedure or code to that point.

Figure 5-6 illustrates a typical source program listing and describes the various pieces of information.

```
 1   1   1:D   1 {$L+}
 2   1   1:D   1 program example;
 3   1   1:D   1 const ten = 1Ø;
 4   1   1:D   1 var i,
 5   1   1:D   1      j: integer;
 6   1   1:D   3      a: array [Ø..5] of integer;
 7   1   1:D   9
 8  1Ø   1:D   1 segment procedure segproc;
 9  1Ø   1:D   1 var localint: integer;
1Ø  1Ø   1:Ø   Ø begin
11  1Ø   1:1   Ø   localint:=i;
12  1Ø   1:Ø   3 end; {segproc}
13  1Ø   1:Ø   6
14   1   2:D   1 procedure proc;
15   1   2:D   1 var localreal: real;
16   1   2:Ø   Ø begin
17   1   2:1   Ø   localreal:=4.5;
18   1   2:Ø   7 end; {proc}
19   1   2:Ø  1Ø
2Ø   1   1:Ø   Ø begin
21   1   1:1   Ø   proc;
22   1   1:1   6   segproc;
23   1   1:Ø  17 end.

 ^   ^   ^ ^   ^      ^
 |   |   | |   |      |
 |   |   | |   |      --------------->  Statement.
 |   |   | |   |
 |   |   | |   -------------------->  In declaration part, number of
 |   |   | |                          words used by declarations.
 |   |   | |
 |   |   | |                          In statement part, byte offset of
 |   |   | |                          program counter relative to start
 |   |   | |                          of procedure.
 |   |   | |
 |   |   | ------------------------>  Declaration flag (D) or nesting
 |   |   |                            level indicator.
 |   |   |
 |   |   -------------------------->  Procedure number.
 |   |
 |   ------------------------------>  Segment procedure number.
 |
 ---------------------------------->  Line number.
```

Figure 5-6. Source Program Listing.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

6. UTILITIES

This chapter describes the various programs available as part of the III.Ø Operating System. Each utility is described and execution instructions are given. The utility capabilies explained in this chapter are listed below.

- SETUP
- BOOTER
- BOOTMAKE
- COPYDUPDIR
- MARKDUPDIR
- LIBRARIAN
- LIBRARY MAP
- LINKER
- P-CODE DISASSEMBLER
- CALCULATOR
- GOTOXY PROCEDURE BINDER
- AUTOMATIC EXECUTION
- FORMATTING FLOPPY DISKS
- FORMATTING WINCHESTER DISKS
- PATCH
- DEBUGGER
- COPY

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

6.1 SETUP

Information about the user's system configuration is kept in a file called SYSTEM.MISCINFO. The user inspects or modifies this file, using the SETUP program. During system initialization, SYSTEM.MISCINFO is read into memory. From there, it is accessed by many parts of the Western Digital UCSD Pascal III.Ø Operating System, particularly by the Screen-Oriented Editor.

Much of the information in this file must be "set up" by the user to conform to the specific hardware configuration and particular needs. Most information pertains to the terminal, keyboard, and disk drives although some miscellaneous information must also be established for any particular configuration.

SETUP is run by typing an X for execute from the system main command line, then the file name SETUP, followed by a carriage return, is entered in response to the file name prompt.

Within SETUP are three levels of prompt selections; the commands associated with each of these levels are shown below.

```
SETUP: C(HANGE T(EACH H(ELP Q(UIT
CHANGE: D(ISPLAYED S(INGLE P(ROMPTED R(ADIX H(ELP Q(UIT
QUIT: D(ISK OR M(EMORY UPDATE, R(ETURN, H(ELP, E(XIT.
```

6.1.1 SETUP Commands

On entry to the program, the user sees the prompt line:

```
SETUP: C(HANGE T(EACH H(ELP Q(UIT
```

If C is entered, the command line for C(HANGE appears. If Q is entered, the command line for Q(UIT appears. If T (for T(EACH) is entered, an explanation of how to use the program appears on the screen. If an H (for H(ELP) is entered, an explanation of what the other commands do is listed. The screen display is shown below:

```
 ----------------------------------------------------------------------
|        C(HANGE) ALLOWS YOU TO CHANGE OR EXAMINE                      |
|            THE VARIOUS PIECES OF INFORMATION THE                     |
|            SYSTEM HAS ABOUT YOUR HARDWARE CONFIGURATION              |
|        T(EACH) TEACHES YOU ABOUT HOW TO USE THIS PROGRAM             |
|        Q(UIT) ALLOWS YOU TO MAKE YOUR CHANGES PERMANENT              |
|            AND LEAVE THIS PROGRAM                                    |
|                                                                      |
|        PRESS RETURN TO CONTINUE                                      |
 ----------------------------------------------------------------------
```

If return is pressed, the SETUP command line reappears.

If the SETUP program is not on the disk, the following message appears:

```
 ----------------------------------------------------------------------
|       no file setup.CODE                                             |
 ----------------------------------------------------------------------
```

SETUP does not tell the system how to do random cursor addressing on the user's terminal. If this feature is required for the user's hardware configuration, information on setting random cursor addressing can be found in Section 6.10, GOTOXY Procedure Binder.

6.1.2 CHANGE Commands

The C(HANGE command line is shown below followed by a brief explanation of each command. The current number base is shown also.

```
 ----------------------------------------------------------------------
|                          *CURRENT NUMBER BASE IS DECIMAL*            |
|   C(HANGE: D(ISPLAYED,S(INGLE,P(ROMPTED,R(ADIX,H(ELP,Q(UIT.          |
 ----------------------------------------------------------------------
```

Entering D (for D(ISPLAYED) displays on the screen all the fields within the SYSTEM.MISCINFO file which can be modified by the user. After entering the D(ISPLAYED option, the user may inspect or modify an individual field by responding with an I (for I(NSPECT) and then entering the appropriate field number.

The fields are listed by number and grouped into two categories: Terminal Group and General Group.

Entering S (for S(INGLE), asks for the name of the field to be changed. When the name is entered, the program responds with the field name, shows the current value, and asks if the field is to be changed ("Y", "N", or "!").

Entering P (for P(ROMPTED) displays each field sequentially on the screen showing the field name, current value, and asking if the field should be changed.

Entering R (for R(ADIX) shows the current number base and allows the user to change the number base to either of the other number bases (three number bases are offered as options: decimal, octal, and hexadecimal). Each time the SETUP program is run, the default radix is decimal. The radix is the number base that the user chooses to use to enter any new value for a field. However, the field values, when displayed, are shown in all three number bases.

Entering H (for H(ELP) from the C(HANGE command line lists the other C(HANGE commands and gives a brief explanation of them as shown below:

```
--------------------------------------------------------------------------
|       D(ISPLAYED: SHOWS ALL FIELDS & ALLOWS CHANGES.                   |
|       S(INGLE: EXAMINES 1 VALUE BY NAME                                |
|       P(ROMPTED: ALL FIELDS SEQUENTIALLY                               |
|       R(ADIX: CHANGES THE ASSUMED RADIX                                |
|            FROM DECIMAL TO EITHER OCTAL OR HEXADECIMAL                 |
|                                                                        |
|       PRESS RETURN TO CONTINUE.                                        |
--------------------------------------------------------------------------
```

Pressing return causes the C(HANGE command line to reappear.

Entering Q (for Q(UIT) causes the SETUP command line to appear on the screen.

6.1.3 QUIT Commands

The Q(UIT command line is shown below with a brief explanation of each command.

```
--------------------------------------------------------------------------
|     Q(UIT: D(ISK OR M(EMORY UPDATE, R(ETURN, H(ELP, E(XIT              |
--------------------------------------------------------------------------
```

The D(ISK option allows the user to write a current configuration to the SYSTEM.MISCINFO file or to a NEW.MISCINFO file. Writing directly to the SYSTEM.MISCINFO file provides immediate implementation on rebooting the system. If the system is not rebooted, the SYSTEM.MISCINFO file remains as it was until the next reboot. If the information is written as a NEW.MISCINFO file, that file may later be changed to the SYSTEM.MISCINFO file.

Entering M (for M(EMORY UPDATE) simply updates the version of the SYSTEM. MISCINFO file in memory (not on disk) with the current configuration. On rebooting the system, that information is lost because it was not stored on disk.

Entering R (for R(ETURN) causes the main SETUP command line to appear on the screen.

Entering H (for H(ELP) lists the Q(UIT commands and gives a brief explanation of each as shown below.

```
------------------------------------------------------------------------
|     D(ISK UPDATE PUTS THE CURRENT SETUP ON DISK                        |
|     AS FILE "NEW.MISCINFO"                                             |
|     OR AS THE FILE "SYSTEM.MISCINFO"                                   |
|     M(EMORY UPDATE CHANGES THE SETUP IN MEMORY                         |
|          UNTIL NEXT SYSTEM INITIALIZATION                              |
|     R(ETURN TAKES YOU BACK INTO SETUP                                  |
|          IF YOU'RE NOT DONE                                            |
|     E(XIT TERMINATES THIS PROGRAM                                      |
|                                                                        |
|     PRESS RETURN TO CONTINUE                                           |
------------------------------------------------------------------------
```

If return is pressed, the Q(UIT command line reappears.

Entering E (for E(XIT) terminates the program returning control to the operating system.

6.1.4 List of Fields in SETUP

Figure 6-1. lists the fields that can be changed through the SETUP program. The figure is laid out as the fields appear on the terminal screen when the D(isplayed option is exercised. The sections following this figure describe the various fields indiviually.

```
                    *********  TERMINAL GROUP ******

1   HAS NO INTERRUPTS                     2  HAS LOWER CASE
3   HAS RANDOM CURSOR ADDRESSING          4  HAS SLOW TERMINAL
5   PREFIXED[MOVE CURSOR UP]              6  PREFIXED[MOVE CURSOR RIGHT]
7   PREFIXED[ERASE TO END OF LINE]        8  PREFIXED[ERASE TO END OF SCREEN]
9   PREFIXED[MOVE CURSOR HOME]            10 PREFIXED[DELETE CHARACTER]
11  PREFIXED[ERASE SCREEN]                12 PREFIXED[ERASE LINE]
13  PREFIXED[NON-PRINTING CHARACTER]      14 PREFIXED[KEY FOR MOVING CURSOR LEFT]
15  PREFIXED[KEY FOR MOVING CURSOR UP]    16 PREFIXED[KEY FOR MOVING CURSOR DOWN]
17  PREFIXED[KEYFORMOVING CURSOR RIGHT]
                                          18 PREFIXED[KEY FOR STOP]
19  PREFIXED[KEY FOR FLUSH]               20 PREFIXED[KEY TO END FILE]
21  PREFIXED[EDITOR 'ESCAPE' KEY]         22 PREFIXED[KEY TO DELETE LINE]
23  PREFIXED[KEY TO DELETE CHARACTER]     24 PREFIXED[EDITOR "ACCEPT" KEY]
25  SCREEN HEIGHT                         26 SCREEN WIDTH

               ENTER C(ONTINUE, I(NSPECT OR E(XIT

                    *********  TERMINAL GROUP ******

27  VERTICAL DELAY CHARACTER              28 LEAD-IN TO SCREEN
29  MOVE CURSOR HOME                      30 ERASE TO END OF SCREEN
31  ERASE TO END OF LINE                  32 MOVE CURSOR RIGHT
33  MOVE CURSOR UP                        34 BACKSPACE
35  ERASE LINE                            36 ERASE SCREEN
37  KEY TO MOVE CURSOR UP                 38 KEY TO MOVE CURSOR DOWN
39  KEY TO MOVE CURSOR LEFT               40 KEY TO MOVE CURSOR RIGHT
41  KEY TO END FILE                       42 KEY FOR FLUSH
43  KEY FOR BREAK                         44 KEY FOR STOP
45  KEY TO BACK SPACE                     46 NON-PRINTING CHARACTER
47  KEY TO DELETE LINE                    48 EDITOR "ESCAPE" KEY
49  LEAD-IN CHAR FROM KEYBOARD            50 EDITOR "ACCEPT" KEY
51  KEY TO DELETE CHARACTER               52 VERTICAL MOVE DELAY

               ENTER C(ONTINUE, I(NSPECT OR E(XIT

                    *********  GENERAL GROUP ******

53  DISK SEEK RATE                        54 DISK READ RATE
55  DISK WRITE RATE                       56 XON/XOFF PROTOCOL
57  BAUDRATE                              58 CLOCK VALUE
59  HAS CLOCK                             60 MENU DRIVEN
61  TRANSPARENT                           62 KEY TO EMPTY QUEUE
63  MAX SERIAL PORTS
```

Figure 6-1. SETUP Fields.

6.1.5 Miscellaneous Information

XON/XOFF PROTOCOL FIELD (GENERAL GROUP - 56)

The I/O drivers for the serial ports support XON/XOFF protocol which uses XON (Control-Q) and XOFF (Control-S) to regulate the speed at which serial data streams are sent so that queue overflow (because of a mismatch of transmitter and receiver speeds) does not cause a loss of data. When the XOFF character is received, a serial transmitter pauses during data transmission until an XON character is received, thus regulating data flow.

The transparent mode is enabled by setting the TRANSPARENT field to TRUE. (See the subsection describing transparent in this section.)

Table 6-1. summarizes the effect of the various configurations of transparent mode and XON/XOFF protocol.

Table 6-1. Transparent Mode and XON/XOFF Protocol.

Standard Configuration Prior to H2 Release.

Transparent	OFF
XON/XOFF	OFF

No Parity Bit Stripping and No Special Character Recognition.

Transparent	ON
XON/XOFF	OFF

XON (Control-Q) and XOFF (Control-S) Transmitted and Interpreted to Prevent Queue Overflow.

Transparent	OFF
XON/XOFF	ON

XON (Control-Q) and XOFF (Control-S) Transmitted and Interpreted to Prevent Queue Overflow. No Parity Bit Stripping and No Special Character Recognition Except for the End-of-File Character. This Mode is Intended for Use During Machine-to-Machine Communications.

Transparent	ON
XON/XOFF	ON

BAUDRATE FIELD (GENERAL GROUP - 57)

This field allows the user to set the baud rate for the serial ports. When this field is accessed for change, the following choices appear.

```
1 BAUDRATE[B]
2 BAUDRATE[C]
3 BAUDRATE[D]
4 BAUDRATE[E]
5 BAUDRATE[F]
6 BAUDRATE[G]

ENTER NUMBER 1..6 OR EXIT
```

The letters in the above list represent the serial ports. After the number for the appropriate port is entered, the current baud rate for that port is displayed. If a Y is answered to the CHANGE THE FIELD PROMPT, the following list of baud rates is displayed.

15) 19.2K	11) 3600	7) 600	3) 134.5
14) 9600	10) 2400	6) 300	2) 110
13) 7200	9) 1800	5) 200	1) 75
12) 4800	8) 1200	4) 150	0) 50

The number of the desired baud rate for that port is entered. BAUDRATE becomes effective at boot, unit clear statements, and Filer volume listings.

CLOCK VALUE FIELD (GENERAL GROUP - 58)

The CLOCK VALUE field allows the clock tick rates to be changed to tick from once every second to once each 1/400 second. The faster the tick rate, the more processor overhead for handling the tick interrupt. The display below appears when the field is to be changed.

```
FIELD NAME IS CLOCK VALUE

OCTAL          DECIMAL   HEXADECIMAL
 17              15          F
     SHOWN IN DECIMAL
```

```
-------------------------------------
| SET                  CORRESPONDING|
| CLOCK VALUE........TICK-RATE      |
|-----------------------------------|
|        1...........none           |
|        3...........1     SEC.     |
|        5...........1/2   SEC.     |
|        7...........1/10  SEC.     |
|        9...........1/20  SEC.     |
|       11...........1/50  SEC.     |
|       13...........1/100 SEC.     |
|       15...........1/400 SEC.     |
-------------------------------------
```

HAS CLOCK FIELD (GENERAL GROUP - 59)

Should be set to TRUE for the ME1600 Series machines which have real-time clocks. For the SB1600 Series machines, setting this option to TRUE enhances system performance by reducing directory reads; however, problems can result if diskettes are swapped in a drive without ensuring the new directory is read in.

MENU DRIVEN FIELD (GENERAL GROUP - 60)

This field allows the user to tailor the operating system to a set of prompts (or menus) in order to make the operating system appear less complex to the end user. (See Section 7.6.3) for additional explanation.)

This field is set to FALSE by default.

TRANSPARENT FIELD (GENERAL GROUP - 61)

When transparent mode is enabled (set to TRUE), no special character recognition interpretation is performed by the operating system. This option assures that interpretation of control characters in a data stream is not performed if this interpretation is not required. In addition, the speed of serial I/O is enhanced because special character checking is not needed.

Under normal conditions for the console unit, the ASCII control characters -- Control-C (ETX), Control-F (flush), Control-S (start/stop), Control-@ (break), and Control-M (carriage return) -- are interpreted by the serial I/O drivers. In addition, on serial input, the high order bit is stripped as some terminals set it. For remote serial communication of binary data, special character interpretation is a drawback.

| NOTE |

In transparent mode, no end-of-file character interpretation is done; therefore, programs may not read until the end of the file.

KEY TO EMPTY QUEUE FIELD (GENERAL GROUP - 62)

This field allows the keyboard type-ahead queue to be emptied. The default value for the key is set to Control-D. This option is useful when erroneous type-ahead has been entered, and the user wishes to remove it before the system can respond to it.

MAX SERIAL PORTS FIELD (GENERAL GROUP - 63)

The MAX SERIAL PORTS field defines the maximum number of serial ports allowed for a given system. The value can be set from Ø to 8. Normally, this field is set to Ø, which causes the default number of serial ports for the system type to be used as the maximum number. The number of serial ports for 16ØØ systems is shown below.

SB16ØØ = 2
ME16ØØ = 4 or 8 (If two serial cards are used in an ME16ØØ system, 8 serial ports are available.)

The user may change the setting depending on the number of ports to be used. Approximately 12Ø words of memory are required per port. Therefore, if the system has four serial ports but only two are being used, 24Ø words of space could be saved by setting this field to 2 (instead of using the default Ø which would show 4 ports). Only when memory space is limited should this field be nonzero.

6.1.6 General Terminal Information

HAS NO INTERRUPTS

This field allows the machine to run in noninterrupt mode. If the machine is having interrupt problems, this field can be changed from FALSE (the default setting) to TRUE in order to debug the system. If this field is not set to FALSE, type ahead does not function because it depends on interrupts being enabled. Therefore, the recommended value is FALSE.

HAS LOWER CASE

If TRUE, the terminal has lower case; otherwise, FALSE.

HAS RANDOM CURSOR ADDRESSING

If TRUE, the terminal has random cursor addressing; otherwise, FALSE. This type of addressing applies only to video terminals.

HAS SLOW TERMINAL

If TRUE, the terminal has a baud rate of 6ØØ or less; otherwise, FALSE. When TRUE, the system issues abbreviated prompt lines and messages.

NONPRINTING CHARACTER

Any printing character may be entered here to indicate the character that should be displayed to indicate the presence of a nonprinting character. The suggested character is an ASCII "?".

SCREEN HEIGHT

Enter the number of lines displayed on the screen of a video terminal. The screen height is usually 24. Otherwise, enter Ø for hard-copy terminal or for one in which paging is not appropriate.

SCREEN WIDTH

Enter the number of horizontal characters displayed on a line of a video terminal. The screen width is usually 8Ø characters (Ø-79). Otherwise, enter Ø for a hard-copy terminal.

VERTICAL DELAY CHARACTER

This key is the pad character output after a slow terminal operation (such as home or clearscreen). The default value is NUL=Ø.

VERTICAL MOVE DELAY

Enter the number of VERTICAL DELAY CHARACTERs to send after a vertical cursor move. The characters are sent after a carriage return, ERASE TO END OF LINE, ERASE TO END OF SCREEN, and MOVE CURSOR UP. The maximum value this field can obtain is 41. Many types of terminals require a delay after certain cursor movements to enable the terminal to complete the movement before the next character is sent.

6.1.7 Control Key Information

Some keyboards generate two codes when a single key is typed. That capability is indicated according to the following format:

```
PREFIXED[<fieldname>]        TRUE
```

The prefix for all such keys must be the same. For example, many keys function as escape keys in addition to their named function. If a user's keyboard had a vector pad that generated the value pairs ESC "U" and ESC "D" for the Up-arrow and Down-arrow keys, respectively, the values below should be entered.

```
KEY FOR MOVING CURSOR UP                 ASCII "U"
KEY FOR MOVING CURSOR DOWN               ASCII "D"
LEAD-IN KEY FOR KEYBOARD                 ESC
PREFIXED[KEY FOR MOVING CURSOR UP]       TRUE
PREFIXED[KEY FOR MOVING CURSOR DOWN]     TRUE
```

The following keys may apply to all terminals.

KEY FOR BREAK

Typing the BREAK key causes the program currently executing to be terminated immediately with a run-time error. This key should be set to something that is difficult to hit accidentally.

KEY TO BACKSPACE

This key specifies the backspace key for the terminal.

KEY TO DELETE CHARACTER

This key removes one character from the current line. It may be typed until nothing is left on the line. The suggested setting is ASCII BS.

KEY TO DELETE LINE

This key causes the current line of input to be erased. The suggested setting is ASCII DEL.

KEY TO END FILE

This key is the console end-of-file character that sets the Boolean function EOF to True. This key designation applies only to INPUT or KEYBOARD files or the unit CONSOLE. The suggested setting is ASCII ETX.

KEY FOR FLUSH

This key is the console output cancel character. When the FLUSH key is pressed, output to the system terminal is undisplayed until FLUSH is pressed again. Processing is uninterrupted although the output is not displayed. The suggested setting is ASCII ACK.

KEY FOR STOP

This key is the console output stop character. When pressed, output to the system terminal ceases. Output resumes where it left off when the key is pressed again. This function is useful for reading data that are being displayed too fast for easy reading. The suggested setting is ASCII DC3.

The following keys are applicable only to video terminals that have selective erase.

EDITOR "ACCEPT" KEY

In the Screen-Oriented Editor, this key is used to accept commands, thus making permanent any action taken. The suggested setting is ASCII ETX.

EDITOR "ESCAPE" KEY

In the Screen-Oriented Editor, this key is used to escape from commands, negating or erasing any action taken. The suggested setting is ASCII ESC.

LEAD-IN CHAR FROM KEYBOARD

This character is the prefix code when two codes are generated for one of the keys described below (namely, keys to move the cursor up, down, and so forth). This character designation applies only to keys that have PREFIXED[<fieldname>] set to TRUE.

KEY TO MOVE CURSOR UP
DOWN
LEFT
RIGHT

These keys are used by the Screen-Oriented Editor for cursor control. If the keyboard has a vector pad, the keys must be set to the value that the vector pad generates. Otherwise, four keys may be chosen in the pattern of a vector pad (for example, "O", ".", "K" and ";") and be assigned the control codes that correspond to them. A prefix character may also be used.

6.1.8 Video Screen Control Characters

The video screen control characters are sent by the computer to the terminal to control the actions of the terminal. The terminal manual gives the appropriate values. If a terminal does not have one of these characters, the field should be set to Ø, unless otherwise directed.

On some terminals, a two-character sequence is required for some functions (for example, ESC plus a character). If the first character for all of the functions is the same, it can be set as the value of the field LEAD-IN TO SCREEN. Then the field PREFIX[<fieldname>] must be set to TRUE for each two-character function.

LEAD-IN TO SCREEN

This character is the prefix code that is used when two character codes must be sent for one of the functions described in the rest of this section. This character designation applies only to functions that have PREFIXED [<FIELDNAME>] set to TRUE.

BACKSPACE

This character causes the cursor to move one space to the left.

ERASE LINE

This character causes the erasure of all characters on the line where the cursor is currently located. The cursor is relocated to the beginning of the line.

ERASE SCREEN

This character erases the entire screen. The cursor is repositioned in the upper left hand corner of the screen.

ERASE TO END OF LINE

This character causes the erasure of all characters from the current position of the cursor to the end of the line. The cursor location is unchanged.

ERASE TO END OF SCREEN

This character causes the erasure of all characters from the current position of the cursor to the end of the screen. The cursor location is unchanged.

MOVE CURSOR HOME

This character causes the cursor to be relocated to "home", which is the upper left hand corner of the screen.

| NOTE |

If the terminal does not have a home character, the field should be set to ASCII CR (carriage return).

MOVE CURSOR UP
RIGHT

These characters cause the cursor to move nondestructively one space in the direction indicated.

6.1.9 Disk Control Information

The SETUP program has three fields, 'DISK SEEK RATE', 'DISK READ RATE', and 'DISK WRITE RATE' that tailor disk accesses. The operating system tailors disk I/O operations by means of these fields. Therefore, the use may configure disk transfer delays and stepping rates of any type of floppy disk drive according to the values set in SETUP. Because of the wide variance of disk drives, this ability is very useful -- full advantage of each type of disk drive is possible. For example, some floppy disk drives have a fast head stepping rate, so the system stepping rate could be modified using SETUP to specify fast step rates.

When the DISK SEEK RATE field is accessed for change, the following choices appear for single- and double-density disks:

```
Ø)  15 ms. + VERIFY
1)  15 ms.
2)  1Ø ms.
3)   6 ms.
4)   3 ms.

NEW VALUE (Ø..4):
```

The 15 ms. rate is the slowest step rate and 3 ms. is the fastest.

For fast drives, a 4) can be selected because the drives have a fast step capability. For slower drives, a 1) or 2) should be selected because the drives have a slow step rate capability. The SYSTEM.MISCINFO file that is shipped with the operating system has a 3) selected, which is a moderately fast step rate and also requests the 1791 Controller not to verify that the SEEK is on the destination track. If a user needs a slower step rate, the file SLOW.MISCINFO contains option Ø) for 15 ms + verify. This file must be changed to SYSTEM.MISCINFO before boot up.

The DISK READ RATE and DISK WRITE RATE fields specify whether or not a delay exists before head load. The choices for DISK READ RATE are listed below:

Ø) NO DELAY
1) DELAY

The choices for DISK WRITE RATE are listed below:

Ø) NO DELAY
1) DELAY

6.1.10 Example

Figure 6-2 presents an example of changing the MENUDRIVEN field in the SYSTEM. MISCINFO file through the SETUP program.

```
Command: E(dit, R(un, F(ile, C(omp, L(ink, X(ecute, D(ebug ?

x
Execute what file? setup<ret>

Setup Miscinfo [H3]

INITIALIZING
SETUP: C(HANGE* , T(EACH* , H(ELP* , Q(UIT*.
c
                          *CURRENT NUMBER BASE IS DECIMAL*
CHANGE: D(ISPLAYED,S(INGLE,P(ROMPTED,R(ADIX,H(ELP,Q(UIT.
s
NAME OF FIELD IS menudriven<ret>
FIELD NAME IS MENUDRIVEN
CURRENT VALUE IS FALSE
 CHANGE THIS FIELD?("Y","N" OR "!")
y
NEW VALUE: t<ret>
   CHANGE THIS FIELD?("Y","N" OR "!")
n

                          *CURRENT NUMBER BASE IS DECIMAL*
CHANGE: D(ISPLAYED,S(INGLE,P(ROMPTED,R(ADIX,H(ELP,Q(UIT.
q
SETUP: C(HANGE , T(EACH , H(ELP , Q(UIT. q<ret>
QUIT: D(ISK OR M(EMORY UPDATE, R(ETURN, H(ELP, E(XIT.
d
SAVE AS?

*NEW.MISCINFO
*SYSTEM.MISCINFO

ENTER N(EW , S(YSTEM , OR E(XIT.
s

QUIT: D(ISK OR M(EMORY UPDATE, R(ETURN, H(ELP, E(XIT.
e
```

{At this point, the main command line reappears. The system must be rebooted in order to load the changed SYSTEM.MISCINFO file.}

Figure 6-2. SETUP Example.

6.2 BOOTMAKE

The BOOTMAKE program is used to configure the bootstrap on all MicroEngine product lines: SB16ØØ, ME16ØØ, and WDØ9ØØ. The H3 operating system interrogates the hardware configuration on which it is running and configures the software system accordingly.

| NOTE |

> In order to achieve a common object, the H3 bootstrap differs from other boots. Therefore, only H3 level bootstraps are operable with the H3 release.

The BOOTMAKE program has two uses: (1) to install a code file as a bootstrap on a floppy or Winchester disk and/or (2) to specify or respecify the system memory size.

The following sequence of actions defines execution of the BOOTMAKE program. (See section 6.13.3 regarding Winchester disks.)

The BOOTMAKE program is executed by typing an X from the system main command line; in response to the prompt for the file to be executed, BOOTMAKE is entered. The following message and prompt then appear.

```
BOOTMAKE version [H3] for floppy and winchester system boots

N(ew boot or C(hange old boot parameters (<return> to quit)?
```

This prompt asks if a new boot is to be installed or if the memory size parameter of an existing boot is to be changed. If an N is entered for a new boot, the following prompt appears.

```
Enter unit Number where boot is to be placed (Ø to exit):
```

Once the unit number is specified, a prompt asks for the name of the code file that contains the boot code. If a Ø is entered in response to the above prompt, the program terminates.

```
Enter code file name ('BOOT' for standard boot):
```

The code file name to be entered is BOOT. The boot must be an H3 or later (compatible) boot because neither BOOTMAKE nor the H3 operating system work with earlier boots.

The next prompt asks for the memory size requirments of the machine. Because the boot cannot automatically determine at boot time how much memory is available on the system, this information must be supplied. If the C(hange option is chosen in the first prompt, the program skips the unit number and code file name prompts.

```
--------------------------------------------------------------------------------
| Enter number of Kbytes of memory for target system.                          |
| Use '128' for ME1600s, SB1600s, or any 128Kbyte system.                      |
| Use '64' for the older 64Kbyte systems.                                      |
| Use '0' to have the system choose automatically 128KB for                    |
| ME1600 and SB1600, and 64KB for older models:                                |
--------------------------------------------------------------------------------
```

This prompt is self-explanatory. An answer of 0 to this prompt permits the boot to choose 128K bytes if the computer is an SB1600 or an ME1600, and to choose 64K bytes for older systems.

After all the questions are answered, the new boot or change is written to the disk specified.

6.3 BOOTER

The BOOTER utility program copies a bootstrap from an exiting disk to a specified unit. On releases prior to the H2 release, when making a copy of a bootable disk, the BOOTER must be used to place a bootstrap on the destination disk. For previous releases, the T(ransfer command in the Filer does not completely copy the bootstrap because track Ø is not normally accessed by the III.Ø Operating System but is reserved for the bootstrap.

On the H2 release of the operating system, however, the Filer T(ransfer command copies the bootstrap to the new volume in a volume-to-volume transfer when copying from a floppy disk to a floppy disk. Although track Ø is copied in volume-to-volume transfer (for example, T(ransfer #4:,#5:), the bootstrap is not copied in other types of transfers. The BOOTER program must be used in those cases.

The BOOTER is run by typing an X for execute from the system main command line. In answer to the file name prompt, BOOTER is entered. When the program executes, the following prompt appears:

```
BOOTER Version[H3]
Bootstrap mover for the Pascal Microengine

To copy a boot from one disk to another, type the unit number
for the destination disk, and the unit number of the source disk.



Unit to write boot to <Ø to exit>:
```

The unit number of the destination disk is entered, followed by a <ret>. The following prompt asking for the source disk unit then appears.

```
Unit this boot is on <Ø to exit>:
```

Once the unit number for the source disk is entered, the BOOTER transfers the bootstrap, and a message appears to indicate that the transfer is completed.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

6.4 DUPLICATE DIRECTORY UTILITIES

Two utilities exist to handle duplicate directories: (1) COPYDUPDIR copies the duplicate directory; and (2) MARKDUPDIR marks a disk that is not currently maintaining a duplicate directory. These utilities are described in the following subsections.

6.4.1 COPYDUPDIR

This program copies the duplicate directory into the primary directory location. The program is entered by typing an X for execute from the system main command line. In answer to the file name prompt, COPYDUPDIR is entered. The following display and prompt appear.

```
-------------------------------------------------------------------
| Duplicate Directory Copier  [H3]                                |
|                                                                 |
| Enter unit # of user's disk <Ø to exit>:                        |
-------------------------------------------------------------------
```

Once the unit number is entered, the following prompt appears:

```
----------------------------------------------------------------------
|Are you sure you want to zap the directory of <volume name>:{blocks 2-5}? |
----------------------------------------------------------------------
```

If a Y for yes is entered, the program copies the duplicate directory onto blocks 2-5 (space for primary directory) and issues a message that the directory was copied.

If an N for no is entered, the following message appears on the screen.

```
---------------------------------------------------------------------
| Directory copy aborted. Type <ret> to exit.                       |
---------------------------------------------------------------------
```

If the disk is not currently maintaining a duplicate directory, the following message appears:

```
---------------------------------------------------------------------
| A duplicate directory is not being maintained on <volume name>:   |
| Type <ret> to exit.                                               |
---------------------------------------------------------------------
```

6.4.2 MARKDUPDIR

This program marks a disk so that it starts maintaining a duplicate directory. The program is entered by typing an X for execute from the system main command line. In answer to the file name prompt, MARKDUPDIR is entered. The following prompt then appears.

```
| Duplicate Directory Marker  [H3]                    |
|                                                     |
| Enter unit # of user's disk <Ø to exit>:            |
```

After the unit number is entered, the program checks to see if a duplicate directory is already being maintained; if not, the following message appears:

```
| A duplicate directory is not being maintained on <volume name>:   |
```

Blocks 6-9 must be free. The program checks for this space and generates the following message if the space is not free.

```
| WARNING! It appears that blocks 6 - 9 are not free for use.     |
|   Are you sure that they are free?                              |
```

If a Y for yes is entered, the program executes the mark, which writes a copy of the primary directory (on blocks 2-5) to blocks 6-9. If a file already resides on blocks 6-9, that file is overwritten. If the user is not sure that the blocks are free, an N for no should be entered in response to the above prompt. In that case, the following prompt appears.

```
| Type <ret> to exit.                                             |
```

After the file is moved to another space or disk, the MARKDUPDIR utility can be executed again.

After the mark is executed, blocks 6-9 can be checked by using the E(xtended command in the File Handler. The extended listing shows where the first file starts. If the first file starts at block 6, or if it starts at block 1Ø but has a four-block unused section at the top, then the disk has not been marked. However, if the first file starts at block 1Ø and no unused blocks occur at the beginning, the disk has been marked.

In the examples below, the disks have not been marked.

```
SYSTEM.PASCAL        1Ø6      1Ø-Jun-82     6     Codefile

                          or

<unused>               4      1Ø-Jun-82     6     Codefile
SYSTEM.PASCAL        1Ø6      1Ø-Jun-82    1Ø     Codefile
```

Below is the directory of a properly marked disk.

```
SYSTEM.PASCAL        1Ø6      1Ø-Jun-82    1Ø    Codefile
```

After the unit number is entered in response to the first prompt, if a duplicate directory is already being maintained, the following message and prompt appear.

```
-------------------------------------------------------------------
| A duplicate directory is allready being maintained on <volume name>:  |
| Mark not done. Type <ret> to exit.                                    |
-------------------------------------------------------------------
```

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

6.5 LIBRARIAN

The Librarian allows the user to link separately compiled Pascal units into a library file. The Librarian is executed by typing an X from the system main command line. In answer to the prompt for the file name, LIBRARY is entered.

The Librarian can be used to link segments of any code file or to add segments to the *SYSTEM.LIBRARY file.

Before adding a segment to the *SYSTEM.LIBRARY file, a new file must be created into which each of the segments to be retained from the original *SYSTEM.LIBRARY file is linked. Segments may then be added by linking from another code file into the new file being created.

NOTE

> The Librarian does not enter a unit into the library unless it contains executable code.

6.5.1 Execution of Librarian

Once the code file LIBRARY has been executed, the following prompt appears:

```
                                  Pascal System Librarian [III.Ø - H3]


  Output code file->
```

The next prompt asks for the link code file.

```
 Link code file->
```

The response is either *SYSTEM.LIBRARY or the user code file from which units are to be linked. Entering an asterisk (*) causes the *SYSTEM.LIBRARY file to be used as the input file.

On specification of the ouput code file and link code file, a ".CODE" suffix is automatically appended to the file name entered (unless the suffix is already present). However, if the file name ends with a period ("."), the Librarian does not add the suffix. For example, the file name "ABC" causes the Librarian to attempt to open "ABC.CODE". The file name "ABC." causes the Librarian to attempt to open "ABC".

The "codekind" (static or vectored) of a code file determines the number of segments a code file can contain and the segment numbers assigned to each segment. Pre-H2 release static code files contain, at most, 16 segments numbered in the range Ø to 15. The H3 release of the III.Ø Operating System supports static code files that can contain 128 segments numbered in the range 128 to 255. Vectored code files can contain, at most, 128 segments numbered in the range 128 to 255.

Once the input (link code file) file name is specified, the program displays the names of all segments currently linked into the input library, each segment number, and the length of each segment in words. Figure 6-3 shows a typical display of the segments in the *SYSTEM.LIBRARY.

Code kind - vectored, Last seg - 132

128-LONGINT	2338	132-DELAYUNIT	159	136-	Ø	14Ø-	Ø
129-SCREENCO	88	133-	Ø	137-	Ø	141-	Ø
13Ø-MENU	32	134-	Ø	138-	Ø	142-	Ø
131-KBDSTUFF	113	135-	Ø	139-	Ø	143-	Ø

Figure 6-3. Display of Segments in Link File.

When the link code file information is displayed, the command prompt line shown below appears near the top of the screen.

```
|Seg# to L(ink + <space>, N(ew file, E(xamine page, M(ultiple link, Q(uit, A(bort|
```

The command options are described in the following paragraphs.

Seg# to Link + <space>

When an L is entered to execute this command, which specifies which segment in the link (input) code file to link into the ouput file, the following prompt appears:

```
| Link seg?                                                              |
```

After a valid segment number is entered, the next prompt is displayed, as shown below:

```
| Seg to link into or <esc>?                                              |
```

This prompt asks for the segment number in the ouput file to which the input segment is to be linked.

After the segment is linked, a display of the output code file is shown. Figure 6-4 shows the display if segment 13Ø from Figure 6-3 has been linked into an output code file.

```
Code file length - 5
Code kind - vectored, Current last seg - 13Ø

128-          Ø   132-       Ø   136-      Ø   14Ø-      Ø
129-          Ø   133-       Ø   137-      Ø   141-      Ø
13Ø-MENU     32   134-       Ø   138-      Ø   142-      Ø
131-          Ø   135-       Ø   139-      Ø   143-      Ø
```

Figure 6-4. Display of Output File After Linking.

N(ew file

This command causes the prompt to reappear for a link (input) code file to be specified. The N(ew file command allows several input files (or parts thereof) to be linked into the output file. This command is executed after all segments from the previous input file have been linked.

If a file name is entered that cannot be opened by the Librarian, a message appears and another file name is requested. Also, if a null file name (a <ret>) is entered, the Librarian restores the link code file to its state before the N(ew file command was invoked.

E(xamine page

Because the display of the link and output code files is limited to one page (16 segments), the E(xamine page command displays the next page of segments. The E(xamine page command first determines which file (L(ink or O(utput) contains the page to be examined.

```
| E(xamine page: L(ink or O(utput file?                            |
```

The next prompt asks for a specific segment number in order to display the appropriate page.

```
| E(xamine page: Seg#?                                              |
```

The segment range is from 128 through 255. If a segment number is entered and that segment does not exist, the following message appears:

```
| There is no page in ___code file containing seg#____.             |
```

The first blank in the above prompt is the code file specified (link or output), and the second blank is the segment number specified to the previous prompt.

If a valid segment number is entered, the segment page containing the specified segment is displayed.

M(ultiple link

The M(ultiple link command allows all segments, or all segments less one, in the link code file to be linked into the output code file at one time.

After an M is typed to invoke the command, the following prompt line appears:

```
| Multiple link: Link A(ll segs, all segs L(ess one or <esc>?        |
```

If an A is entered to link all segments, the following prompts, except the one asking for the segment to exclude, appear. Likewise, if an L is entered to link all segments from the link file, less one, to the output file, the following prompts appear. In that case, the prompt below also appears.

```
|  Multiple link:  Exclude what seg# or <esc>?                      |
```

If, for example, one segment had already been linked into the output file, that segment number would be the segment number entered to exclude from the linkage.

In both cases, the next prompt asks if the segment numbers in the link file are to be preserved in the output file.

```
-----------------------------------------------------------------------------
| Multiple link:  Preserve seg numbers across linkage (y,n,<esc>)?           |
-----------------------------------------------------------------------------
```

The Librarian determines whether or not the segment numbers of the link code segments may be preserved across the linkage. That is, whether or not, for each link segment being linked, the corresponding segment in the output code file is empty. If the segment numbers may not be preserved, a message appears and an option to abort the multiple linkage is given. If the user chooses to continue, the Librarian links segments increasingly from the first available segment in the output code file.

If the segment numbers can be preserved, the user has the option to do so. If the segment numbers are to be preserved, the Librarian links each segment of the link code to the same segment in the output code. If the segment numbers are not be to preserved, the segments are linked increasingly from the first available segment in the output code file.

NOTE

If the code kind of the output file has not been determined at the time the M(ultiple link command is invoked, the Librarian sets it to the code kind of the link code file.

Once all prompts are answered, the linking process begins and the following message appears which shows the segment numbers as they are linked. Once all segments are linked, the second message appears.

```
-----------------------------------------------------------------------------
| Multiple link: Linking seg# 128...129...130...                             |
| Multiple link: Linkage complete                                            |
-----------------------------------------------------------------------------
```

Q(uit

The Q(uit command terminates the Librarian and causes the output file to be written. A prompt appears asking for a copyright notice; the copyright notice is added to the output code file.

A(bort

The A(bort command terminates the Librarian without writing the output code file.

Once the Q(uit or A(bort command is invoked, the system main command line reappears. If the *SYSTEM.LIBRARY file has been used as the link code file and the output code file is to be a new *SYSTEM.LIBRARY file, the old version should either be removed, or the name changed if it resides on the same disk as the new file. The new file should be given a different name until the old version is changed or removed; the new file name should then be changed to *SYSTEM.LIBRARY in order to be used.

6.5.2 Error Modification

The H3 III.Ø Operating System Librarian displays error messages when a specified command or response cannot be completed. All input data are checked thoroughly to ensure that the data make sense in the current context.

6.6 LIBRARY MAP

The LIBMAP program produces a map of a library or code file and lists the linker information maintained for each segment of the file. (See 6.7, LINKER for additional information.) The program is entered by typing an X from the system main command line. In response to the file name prompt, LIBMAP is entered. The following display and prompt appear asking for a library file name.

```
| Library map utility  [III.Ø H3]                                 |
| enter library name:                                             |
```

An asterisk (*) entered in response to the above prompt indicates *SYSTEM.LIBRARY. The ".CODE" suffix may be suppressed when requesting a library or file other than *SYSTEM.LIBRARY by appending a period to the full file name. Examples are given below.

Entered by User	To Reference the File
*	*SYSTEM.LIBRARY
DIGITAL	DIGITAL.CODE
DIG.LIBRARY.	DIG.LIBRARY

LIBMAP is usually used to list library definitions. The following prompt appears for this selection.

```
| list linker info table (Y/N)?                                   |
```

The listing may also include intralibrary symbol references. The prompt for this selection is given below:

```
| list referenced items (Y/N)?                                    |
```

The next prompt asks for the output file name.

```
| map output file name:                                           |
```

If an extra period is not added at the end of the file name (and the file is on a block-structured device), LIBMAP automatically appends the suffix ".TEXT" to the file name. The LIBMAP program supports static and vectored 128-segment code files.

Several libraries may be mapped at one time. After the first mapping is completed, another library files may be mapped if desired. Typing a <ret> in response to this prompt terminates the program; the system main command line reappears at the top of the screen.

Figure 6-5 shows an example of a map output file for the system library (*SYSTEM.LIBRARY).

```
LIBRARY MAP FOR *SYSTEM.LIBRARY


Segment #128:  LONGINT   library unit


  type fakefib = integer;
       fakedecmax = integer[36];

  procedure fwritedec (var ff: fakefib; faked: fakedecmax; rleng: integer);
  procedure freaddec (var ff: fakefib; var fakeli: integer; l: integer);
  procedure decops (dummy: integer);



    LONGINT  unit byte reference (2 times)

     ----------------------------------------------------------------------

Segment #129:  SCREENCO  library unit


  type months = Ø..12;
       days   = Ø..31;
       years  = Ø..99;

  procedure home;
  procedure cleareos;
  procedure cleareol;
  procedure date (var m: months; var d: days; var  y: years);
  function  screenwidth: integer;
  function  screenheight: integer;



    SCREENCO unit byte reference (Ø times)

     ----------------------------------------------------------------------

Segment #13Ø:  MENU      library unit


  procedure chain (title: string);
  procedure menuenable;
  procedure menudisable;



    MENU     unit byte reference (Ø times)
```

Figure 6-5. Map Output File - *SYSTEM.LIBRARY. (Page 1 of 2)

Segment #131:   KBDSTUFF   library unit

```
procedure kbdbatch(funit: integer; kstring: string);


  KBDSTUFF unit byte reference (Ø times)
```

Segment #132:   DELAYUNI   library unit

```
type

  semptr = ^semaphore;

  timeobject = record
                 delay_sem   : semptr;       { semaphore to signal to awaken }
                 timed_out   : ^boolean;     { set true if timed out         }
                 time_outhi  : integer;      { time to be awaken             }
                 time_outlo  : integer;      { time to be awaken             }
                 time_link   : ^timeobject   { points to next clocknode      }
               end;

procedure time_out_delay (seconds: integer; var delaynode : timeobject;
                          var sem : semaphore; var timeout : boolean);

procedure delay (seconds: integer);

procedure cancel_time_out (var self : timeobject);


  DELAYUNI unit byte reference (Ø times)
```

Figure 6-5. Map Output File - *SYSTEM.LIBRARY. (Page 2 of 2)

6.7 LINKER

The III.Ø Operating System Linker and Library facilities allow the user to:

- Create libraries of frequently used subroutines.
- Divide lengthy programs into modules that can be compiled separately.

The precompiled files are then linked together and can be executed as one file. In most cases, the precompiled files reside in the file *SYSTEM.LIBRARY and are combined with the current work file.

The Linker is usually invoked automatically when a user program is executed. In some cases, the user invokes the Linker explicitly.

During the linking process, the Linker reports all segments being linked and all external routines being copied into an output file. If the user chooses, the Linker produces a map file that contains information relevant to the linking process.

6.7.1 Source Program Organization

In writing programs that use precompiled routines or subprograms, the user must use them in the calling program. The Compiler then informs the system that linking is required before execution.

The Linker can also link UNITs (groups of routines used together to performs a common task) into the program. Any files that reference UNITs and have not yet been linked may be compiled and saved, but must be linked together before execution.

6.7.2 Using the Linker

The Linker is invoked by typing an L for L(inker or an R for R(un from the outer level of commands. In the following cases, the Linker must be explicitly invoked:

- If the file into which the routines are to be linked is not the work file.
- If the external routines to be linked reside in library files other than *SYSTEM.LIBRARY.

When an L is entered, the Linker responds with the following prompt.

```
| Host file?                                                          |
```

The host file is the file into which the routines or UNITs are to be linked. If the work file is to be used, an asterisk and a <ret> are entered rather than a file name. Any file name entered is automatically appended with the suffix ".CODE" by the Linker. The next prompt asks for the name of the libary file in which the UNITs or external routines reside.

```
|  Lib file? <code file identifier> <ret>                             |
```

Up to eight library file names may be entered. Typing an asterisk and a <ret> cause the Linker to reference the *SYSTEM.LIBRARY file. A message reporting each library file successfully opened appears on the screen; an example is shown below:

```
| Lib file?  *<ret>                                                   |
|                                                                     |
| Opening *SYSTEM.LIBRARY                                             |
```

When all library file names have been entered, the user must type a <ret> to proceed. The next prompt is as follows:

```
| Map file? <file identifier <ret>                                    |
```

The Linker writes the map file to the file requested. The map file contains information relevant to the linking process. Unless a period is the last letter of the file name, the Linker automatically appends the suffix ".TEXT." to the file name.

If a <ret> alone is entered, the map file option is not selected.

After all the segments required for linking have been read, a prompt requesting a destination file name for the linked code output appears on the screen.

```
| Output file?                                                        |
```

Often the destination file is the same as the host file. After a <ret> is typed following the output file name, linking begins.

If a <ret> alone is entered, the output file is placed on the work file, *SYSTEM>.WRK.CODE.

During the linking process, a progress report appears on the console. All segments being linked and all external routines being copied into the output file are reported.

If any required segments or routines are missing or undefined, linking is aborted. One of the following messages appears on the screen.

```
Unit <identifier> undefined
Proc <identifier> undefined
Func <identifier> undefined
Global <identifier> undefined
Public <identifier> undefined
```

When using the Run command, if the program in the work file contains EXTERNAL declarations or uses UNITs, the Linker is automatically invoked after the Compiler. The Linker searches the *SYSTEM.LIBRARY file for the routines or UNITs specified and attempts to link them.

If the routine or UNIT is not in *SYSTEM.LIBRARY, the Linker responds with an appropriate message from the previous list.

6.7.3 Conventions and Implementation

A code file may contain up to 128 segments. Block Ø of the program code file contains information regarding name, kind, relative address, and the length of each code segment. This information is called the segtable. The following Pascal data structures describe the segment table format, which allows vectored 128-segment code files.

```
const
   firstsysseg  = Ø;
   maxsysseg    = 127;
   firstuserseg = 128;
   maxuserseg   = 255;
   maxsubseg    = 15;
```

firstsysseg -- The segment # assigned to the first segment in system level (static .. $U-) program . (firstsysseg is the minimum segment number in system level programs.)

maxsysseg -- The maximum segment # allowed in a system level program.

firstuserseg -- The segment # (minimum) assigned to the first segment in user (vectored .. $U+) programs.

maxuserseg -- The maximum segment # allowed in a user program.

maxsubseg -- The maximum index in a segment table page (from Ø).

```
type
  segsubrange = Ø..maxsubseg;
  segrange = firstsysseg..maxuserseg;
  segkinds = (linked, hostseg, segproc, unitseg, seprtseg);
  codeknds = (static, vectored);
  alpha = packed array [Ø..7] of char;

  segpage = record
              diskinfo     : array [segsubrange] of
                               record
                                 codeleng,
                                 codeaddr : integer
                               end;
              segname      : array [segsubrange] of alpha;
              segkind      : array [segsubrange] of segkinds;
              textaddr     : array [segsubrange] of integer;
              seginfo      : array [segsubrange] of
                               packed record
                                 segnum : segrange;
                                 codeversion : Ø..255
                               end;

              notice       : string[79];
              codekind     : codeknds;
              lastseg      : segrange;
              lastcodeblk  : integer;
              filler       : packed array [1..138] of char
            end { segpage };
```

segpage -- This record is the modified format of segment table "pages" implemented to support 128 segment programs. The record is identical to the pre H2 "16 segment maximum" segment table except for the additions of 'codekind', 'lastseg' and 'lastcodeblk'. This similarity is necessary to allow upward compatibility of older code files.

The segment table portions of code files consists of 1 to 8 of these segment table pages. Each page contains information on 16 segments.

|NOTE|

Pre-H2 code files require only one page because they were restricted to a maximum of 16 segments.

The first segment table page is stored in the first block of all code files. The format of the contents of the complete segment table is described by the new fields 'codekind', 'lastseg', and 'lastcodeblk'. These three fields are necessary only in the first page but are repeated in subsequent pages for the sake of consistency.

A 'codekind' value of 'static' specifies that the code file contains either a pre-H2 program or a system level ($U-) program consisting of 1 to 128 segments (numbered in the range Ø to 127 -- "firstsysseg..maxsysseg").

A 'codekind' value of 'vectored' specifies that the code file contains a user program consisting of 1 to 128 segments (numbered in the range of 128 to 255 -- "firstuserseg..maxuserseg").

The values 'lastseg' and 'lastcodeblk' are used to define the location and number of remaining segment table pages. The value of 'lastseg' is the largest segment # of the segments contained inthe code file. If this value is in the range of Ø..15 or 128..143, then all segment table information is contained in the first page and the value of 'lastcodeblk' is extraneous. If this value is in the range 16..127 or 143..255, then more segment table pages (at least 1 more) exist in the code file. These remaining pages are stored sequentially

starting at the first block beyond the block specified by 'lastcodeblk'. ('lastcodeblk' is the block number of the last block in the code file containing code. For example, blocks 1 through 'lastcodeblk' are the "code portion" of the code file.)

The 'lastseg' value absolutely determines the number of segment table pages in the file. A code file contains only as many segment table pages as are necessary. For example, if 'lastseg' is 188, segment table pages 5 through 8 are not present. Also, each segment table page can contain only a specific group of 16 segments. That is, page 5 contains segment information for segments 64 through 79 (static) or 192 through 2Ø7 (vectored) only. Thus, if a code file contains only segments 2 and 94, the segment table consists of 6 pages. Page 1 contains segment information for segment 2; pages 2 through 5 are present but empty; page 6 contains segment information for segment 94, and pages 7 and 8 are not present at all.

|NOTE|

> The Debugger stores breakpoint information in the 'filler' field of page 1 only, regardless of whether or not other pages are present.

Figure 6-6 shows the code file represented.

```
                              Code File
                  ------------------------------------
                / |  segment info for segs Ø..15 or    |
               /  |  segs 128..143.                    |
              /   |....................................|
    block Ø  <    |  codekind                          |
    (page 1)  \   |  lastseg                           |
               \  |  lastcodeblk  ( = N )              |
                \ |                                    |
                  |------------------------------------|
                / |                                    |  \
    block 1    <  |                                    |   \
                \ |                                    |    \
                  |------------------------------------|     C
                  /                                    /     O
                 /                                    /      D
                  |------------------------------------|     E
                / |                                    |    /
    block N    <  |                                    |   /
                \ |                                    |  /
                  |------------------------------------|
                / |                     16 ..  31      |
    block N+1  <  |   seg page for segs     or         |
    (page 2)    \ |                    144 .. 159      |
                  |------------------------------------|
                / |                     32 ..  47      |
    block N+2  <  |   seg page for segs     or         |
    (page 3)    \ |                    16Ø .. 175      |
                  |------------------------------------|
                / |                     48 ..  63      |
    block N+3  <  |   seg page for segs     or         |
    (page 4)    \ |                    176 .. 191      |
                  |------------------------------------|
                / |                     64 ..  79      |
    block N+4  <  |   seg page for segs     or         |
    (page 5)    \ |                    192 .. 2Ø7      |
                  |------------------------------------|
                / |                     8Ø ..  95      |
    block N+5  <  |   seg page for segs     or         |
    (page 6)    \ |                    2Ø8 .. 223      |
                  |------------------------------------|
                / |                     96 .. 111      |
    block N+6  <  |   seg page for segs     or         |
    (page 7)    \ |                    224 .. 239      |
                  |------------------------------------|
                / |                    112 .. 127      |
    block N+7  <  |   seg page for segs     or         |
    (page 8)    \ |                    24Ø .. 255      |
                  |------------------------------------|
```

Figure 6-6. Code File Diagram.

CODELENG gives the length of the segment in words, and CODEADDR gives the block address. A description of SEGKIND follows:

LINKED	A fully executable code segment. Either all external references have been resolved or none were present.
HOSTSEG	The outer block of a Pascal program if the program has external references.
SEGPROC	A Pascal segment procedure.
UNITSEG	A segment that is the result of compiling a unit. It will be linked into a host as a new segment.

If a segment contains unresolved external references, the Compiler generates linker information. This information is a series of variable-length records, one for each UNIT, routine, or variable that is referenced in but is external to the source. The first eight words of each record contain the following:

```
LIENTRY=RECORD
        NAME: ALPHA;
        CASE LITYPE: LITYPES OF
           UNITREF,
           GLOBREF,
           PUBLREF,
           PRIVREF,
           SEPFREF,
           SEPPREF,
           CONSTREF:
             (FORMAT: OPFORMAT;           {format of LIENTRY; name can be}
                                          {BIG BYTE, or WORD}
              NFEFS: INTEGER;             {# of references to lientry name}
                                          {in compiled code segment}
             NWORDS: LCRANGE);            {size of privates in words}
          GLOBDEF:
            (HOMEPROC: PROCRANGE;         {which procedure it is in byte}
              ICOFFSET: ICRANGE;)         {offset in p-code}
           PUBLDEF:
             (BASEOFFSET: LCRANGE);       {compiler-assigned word offset}
           CONSTDEF:
             (CONSTVAL: INTEGER);         {user's defined value}
           EXTPROC, EXTFUNC,
           SEPROC, SEPFUNC:
             (SRCPROC:  PROCRANGE;        {procedure # in source segment}
              NPARAMS: INTEGER);          {# of parameters expected}
           EOFMARK:
             (NEXTBASELC:  LCRANGE);      {private var allocation info}
           END(*lientry*)
```

If the LITYPE is one of the first case variants, a list of code pointers (references) into the code segment follows this portion of the record. Each pointer is the absolute byte address within the code segment of a reference to a variable, UNIT, or routine named in LIENTRY. These are 8-word records; but only the first NREFS are valid.

6.8 P-CODE DISASSEMBLER

The P-code disassembler inputs a UCSD code file and outputs symbolic Pascal code (P-code). The disassembler is helpful to the user in optimizing programs and provides a source of information on the subtleties of the UCSD implementation of Pascal.

6.8.1 Disassembly

The disassembler is invoked by typing an X for execute from the system main command line. In response to the file name prompt, DISASM is entered. The first prompt from the program is for an input code file.

The suffix ".CODE" is assumed and therefore is not required. The code file must be one that has been generated by the Pascal Compiler. If a program USES a UNIT, the disassembly program includes the UNIT only if the code file has been linked.

The next prompt is for an output file for the disassembled output. Any file may be specified.

The user may decide whether or not to take control of the disassembly to disassemble only selected procedures or disassemble all in the file.

The Segment Guide displays the segments in the file by name so that a particular segment can be selected if the user controls disassembly. The Segment Guide is exited by typing Q. Figure 6-7 gives an example of a Pascal program and its disassembly. The P-code disassembler supports static and vectored 128-segment code files.

```
 1  128   1:D     1 (*$L DISASSM.LIST*)
 2  128   1:D     1 PROGRAM DISASSM;
 3  128   1:D     3
 4  128   1:D     3   VAR J,I : INTEGER;
 5  128   1:D     5       BUF : ARRAY[Ø..6] OF INTEGER;
 6  128   1:D    12
 7  128   1:Ø     Ø   BEGIN
 8  128   1:1     Ø     J :=4;
 9  128   1:1     5     I .=J+1;
1Ø  128   1:1    1Ø     BUF[J] :=2ØØ;
11  128   1:Ø    22   END.
```

Sample Pascal Program

```
DATA POOL:      BLOCK # 1   OFFSET IN BLOCK= Ø
offsets given are words from start of segment

  Ø:11ØØ.   |   1DØØ.   |   Ø9ØØ.   |
```

SEGMENT	PROC	OFFSET#	BLOCK # 1 / OFFSET IN BLOCK= 6		HEX CODE
128	1	Ø(ØØØ):	SLDC	6	Ø6
128	1	1(ØØ1):	LSL	Ø	99ØØ
128	1	3(ØØ3):	SPR		D1
128	1	4(ØØ4):	SLDC	4	Ø4
128	1	5(ØØ5):	STL	2	A4Ø2
128	1	7(ØØ7):	SLDL	2	21
128	1	8(ØØ8):	SLDC	1	Ø1
128	1	9(ØØ9):	ADI		A2
128	1	1Ø(ØØA):	STL	1	A4Ø1
128	1	12(ØØC):	LLA	3	84Ø3
128	1	14(ØØE):	SLDL	2	21
128	1	15(ØØF):	SLDC	Ø	ØØ
128	1	16(Ø1Ø):	SLDC	6	Ø6
128	1	17(Ø11):	CHK		CB
128	1	18(Ø12):	IXA	1	D7Ø1
128	1	2Ø(Ø14):	LDCB	2ØØ	8ØCB
128	1	22(Ø16):	STO		C4
128	1	23(Ø17):	RPU	9	96Ø9

Sample Program Disassembled

Figure 6-7. Disassembly Example.

6.9 THE CALCULATOR

The calculator program is entered by typing an X for execute from the system main command line. In response to the file name prompt, CALC is entered. The following prompt appears after the program is executed.

```
 ----------------------------------------------------------------------------
 |      ->                                                                  |
 ----------------------------------------------------------------------------
```

This prompt expects a one-line expression in algebraic form as a response. Up to 25 different variables are available, each with different values assigned using the syntax of the given grammar. Only the first eight letters may be used to distinguish between variables. Variables having a value may be used as constants. The two built-in variables are PI(3.141593) and E(2.718282). No distinction is made between upper and lower case letters.

The Pascal MOD function (represented by '/' in the CALC program) rounds the operands to integers.

| WARNING |

> Because this function uses the Pascal definition of MOD (Jensen and Wirth, p.108), the results obtained may not be as expected.

The operand of the factorial FAC function also is rounded to an integer that must be between 0 and 33, inclusive, or the expression is rejected.

The up arrow is used for exponentiation. The answer is calculated by using e ^ Y LN (X). Therefore, the operand must be positive or the expression is rejected.

The keyword LASTX is assigned the value of the previous correct expression and may be used in the next expression. This capability eliminates the necessity for reentering the same expression.

Angles for the TRIG functions must be in RADIANS. Degree-to-radian conversion is accomplished by RADANGLE = (PI/180) * DEGANGLE.

The calculator program fails on an execution error if an overflow or underflow occurs. If this happens, all user-assigned variables and their values are lost. Type carriage return immediately following the prompt to leave the calculator. Calculator examples are given in Figure 6-8.

```
-> PI
     3.14159

-> E
     2.71828

-> A = (FAC(3)/2)
     3.ØØØØØ

-> 3 + 6
     9.ØØØØØ

-> A + 6
     9.ØØØØØ

-> <RET>    To end the program
```

Figure 6-8. Calculator Examples.

6.10 GOTOXY PROCEDURE BINDER

The GOTOXY binder alters the SYSTEM.PASCAL on the default P(refix disk in order to create and bind into the system (once only) the GOTOXY procedure that enables the system to communicate correctly with a specific video terminal. Only system configurations containing a video terminal need GOTOXY. The coordinates for the upper lefthand corner of the video screen must be X=0, Y=0.

The H3 operating system is configured for the class of terminals including Soroc IQ120 and the Ampex Dialog 80. For users with Volker-Craig model terminals, switch #1 on the back panel of the terminal may be set to ADM-3 mode rather than VK404 mode. In ADM-3 mode, the Volker-Craig terminal emulates a Soroc IQ120 terminal.

The GOTOXY binder is entered by typing an X from system main command line. In response to the file name prompt, BINDER is entered. The program then prompts as below:

```
| Enter name of file with GOTOXY(x,y: integer) procedure:                  |
```

This prompt requests the name of the code file containing the compiled version of the procedure that must be created to suit the needs of the particular installation.

To create "local GOTOXY", examine the example GOTOXY procedure shown in Figure 6-9. It is the procedure for doing GOTOXY cursor addressing for the Soroc IQ120 terminal.

```
{$R-}
PROGRAM EXAMPLE
{GOTOXY FOR SOROC IQ12Ø}
PROCEDURE IQ12ØXY(X,                          {COLUMN NUMBER}
                  Y: INTEGER);                {ROW NUMBER}
VAR P: PACKED ARRAY[Ø..3] OF CHAR;
BEGIN
  IF Y>23 THEN Y:=23
  ELSE IF Y<Ø THEN Y:=Ø;
  IF X>79 THEN X:=79
  ELSE IF X<Ø THEN X:=Ø;
  P[Ø]:=CHR(27);
  P[1]:='=';
  P[2]:=CHR(Y+32);
  P[3]:=CHR(X+32);
  UNITWRITE(1,P,4,,12);
END;
BEGIN   END.
```

Figure 6-9. GOTOXY Procedure for Soroc IQ12Ø Terminal.

In order to create a GOTOXY procedure for a specific terminal, a text editor must be used. If the system does not have a terminal that supports the GOTOXY procedure that was shipped with the system, then the user may not use the Screen-Oriented Editor because it is dependent on a correct GOTOXY procedure. In this case, YALOE, the line-oriented editor must be used to create the new GOTOXY procedure.

Modify the GOTOXY procedure to meet the specifications of the intended terminal, recompile it, and run BINDER on it. GOTOXY must be the only procedure declared within the source program (other than the dummy system main program which should be empty). The GOTOXY procedure must not make use of any STRING or REAL constants. Also, SETUP must be run on the newly produced system code file in order for GOTOXY to work properly.

If the GOTOXY binder gives the error "IO ERROR IN WRITING OUT SEGMENT" an error occurred when it tried to write out the newly bound file. Check to make sure enough room exists on the diskette for the file. On the H3 release at least 1Ø6 blocks of open disk space are required to write out the new operating system. (A space equal to or greater than SYSTEM.PASCAL must be available for this write.)

6.11 AUTOMATIC EXECUTION

If a file named SYSTEM.STARTUP exists on the boot diskette, the file is executed automatically when the system is booted or whenever the system is reinitialized (for example, when I for I(nitialize is typed). Examples of the use of this function are to send characters to initialize the terminal or to automatically start a program in an environment where the user only knows how to use a particular program, not how to use the III.Ø Operating System.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

6.12 FORMATTING FLOPPY DISKS

To format a floppy disk, the FORMAT program is executed by typing X from the system main command line. In response to the file name prompt, FORMAT is entered. The following message and prompt appear.

```
| Floppy disk formatter                          Version [H3]    |
|                                                                |
| Unit containing disk to be formatted <Ø to quit>:              |
```

After the unit number is entered, the following prompts appear one at a time. The second one does not appear until the first is answered and so forth. No <ret> is required at the end of the line.

```
| Format single or double density (S or D) ?                     |
| Format single or double sides (S or D) ?                       |
| Format all tracks ( Y or N ) ?                                 |
| Verify ( Y or N ) ?                                            |
```

After responding to all the prompts, formatting begins and a running count of the track number is displayed at the top of the screen.

```
| Formatting     <track number>                                  |
| Verifying      <track number>                                  |
```

If an N is entered for the "format all tracks" prompt above, the following two prompts appear, one after the other when answered.

```
| Enter starting track number                                    |
| Enter final track number                                       |
```

The two prompts above must be followed by a <ret>.

Any errors encountered are displayed in the bottom half of the screen.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

6.13 CONFIGURING WINCHESTER DISKS

The H3 version of the III.Ø Operating System supports Winchester disk drives on the ME16ØØ product line. Because a Winchester disk has considerably more space than a floppy disk, a Winchester disk may be partitioned in multiple volumes. The maximum number of volumes allowed on a Winchester disk is 236.

The H3 operating system provides for the automatic incorporation of a Winchester I/O driver if a Winchester disk exists on the system. The H3 operating system also allows software controlled single-double density selection; however, for an ME16ØØ, this feature requires that the floppy disk controller card be upgraded to hardware level B8 or later. Also, on an H3 Winchester system, a volume-to-volume transfer between two volumes on a Winchester disk unit asks for the new name of the destination volume. This feature prevents file access ambiguity caused by duplicate volume names.

For maximum flexibility in performing the partitioning into volumes, each Winchester disk has a configuration record on track Ø which describes the volume partitioning. A floppy disk may also have a configuration record, but a configuration record would normally only be put on a floppy disk for a special and unusual purpose. If a floppy disk does not contain a config-guration record, the configuration defaults to units 4, 5, and 9-14, as floppy disk volumes with no Winchester volumes.

This general introduction describes how to configure a new Winchester disk. After the general discussion, specific descriptions of the WFORMAT, CONFIGURE, and BOOTMAKE programs follow, with detailed operational steps for configuring a Winchester disk using the programs.

At boot, the ME16ØØ first attempts to boot from any floppy disks on line. If no floppy disks are on line, the system boots from the Winchester disk. If a floppy disk without a boot is on line, the system attempts to boot from it. However, the lack of a boot on the floppy causes booting to fail.

|NOTE|

> Care should be taken to assure that no floppy disks are on line when booting from the Winchester disk is desired.

With the H3 III.Ø Operating System, the volumes on a Winchester disk may be any unit number. The boot unit may, therefore, be any unit number, not only unit #4. When configuring a Winchester disk with the CONFIGURE program, the Winchester disk should not be unit #4. This recommendation is made because floppy disks, which have no configuration records, (almost all floppy disks are in this state) are unit #4. Thus, a conflict can arise if a Winchester is also unit #4; in that case, the system is unable to read the floppy disk. Therefore, to allow the flexibility to read floppy disks, the Winchester disk should not be unit #4 or #5; preferably a unit number in the range 28..255 should be used..

A general outline of steps to configure a new unformatted Winchester disk follows. The steps are explained in more detail in subsequent subsections.

1. Boot the system with the H3 system floppy disk.

2. Execute the WFORMAT program to format the Winchester disk. As part of the formatting operation, the WFORMAT program writes a configuration record to track Ø of the Winchester disk being formatted. The configuration record describes the physical characteristics of the Winchester disk. The following information is the pertinent data for 1Ø and 4ØMB Winchester Disks.

   - 1Ø MB Winchester Disk
     2 heads, 512 cylinders, step rate Ø
     1Ø24 tracks (2*512) each holding 16 blocks
     16K blocks (16*1Ø24) or 8MB of usable data space

   - 4Ø MB Winchester Disk
     8 heads, 512 cylinders, step rate Ø
     4Ø96 tracks (8*512) each holding 16 blocks
     64K blocks (16*4Ø96) or 32MB of usable data space

3. Execute the CONFIGURE program to partition the Winchester disk in multiple volumes. The CONFIGURE program adds this operating system configuration information to the configuration record initially created by the WFORMAT program.

4. Reboot the system so that the newly created configuration record can be read into memory by the III.Ø Operating System.

5. Execute the BOOTMAKE program to install a boot on the Winchester disk. The codefile BOOT is the bootstrap to be placed on the Winchester disk by BOOTMAKE.

6. Transfer all files from the H3 system floppy disk to the lowest numbered Winchester volume.

7. Reboot the system from the Winchester disk — NO floppy disk on line.

To set up a new H3 Winchester-based operating system, an H3 system diskette containing the following codefiles is required:

WFORMAT
CONFIGURE
BOOTMAKE
BOOT

6.13.1 The WFORMAT Program

The WFORMAT program formats a Winchester disk and places a configuration record on track Ø after formatting the disk. This configuration record describes such hardware characteristics as number of heads and cylinders. In addition, the WFORMAT program has the capability to map out bad sectors if one is detected on the Winchester disk media.

To format a Winchester disk, the WFORMAT program must be run first. The following steps describe how to run the WFORMAT program.

1. An X (for execute) is entered from the system main command line. In response to the file name prompt, WFORMAT is entered.

2. The following prompt is displayed:

```
--------------------------------------------------------------------------
|************************************************************************|
|WESTERN DIGITAL WINCHESTER FORMAT PROGRAM H3                            |
|************************************************************************|
|                                                                        |
| CAUTION! THIS PROGRAM WILL DESTROY ANY DATA ALREADY ON                 |
|          THE WINCHESTER IN THE AREA THAT IS REFORMATTED.               |
|                                                                        |
| Do you wish to format the Winchester?                                  |
--------------------------------------------------------------------------
```

   A Y response is entered to format a Winchester disk. If a Y response is entered, the Winchester drives on line are displayed. An N response causes the program to terminate.

3. The program displays the Winchester drives that are on line. If more than one drive is on line, the program asks which drive to format The number of the Winchester drive to format should be entered.

4. If the disk has been configured for an H3 system previously, the WFORMAT program displays a message that a valid configuration record exists on the drive and then asks if it should be retained. This feature is useful in the case where only a particular area on the disk is to be reformatted and the existing configuration is to be retained.

   A Y is entered if the existing configuration record is to be retained.

   An N is entered to generate a new configuration record based on the information to be given in response to program prompts.

5. If the disk did not have a configuration record previously or if the existing record was discarded in step 4, the program prompts for information regarding the drive. The information necessary as pertains to Western Digital Winchester drives currently supported is summarized below for reference.

   - Number of heads
     for 10 M bytes: 2<ret>
     for 40 M bytes: 8<ret>

   - Number of cylinders

     Enter: 512<ret> (all Quantum drives have 512 cylinders)

   The program assumes the number of blocks per track to be 16.

   - Step Rate

     Enter: 0<ret> (all Quantum drives can use a step rate of 0)

6. The WFORMAT program then calculates the interleave pattern and displays it on the screen. The following prompt line also appears:

```
Format E(ntire disk, C(ylinder range, T(rack, Q(uit?
```

   To format the entire Winchester disk, an E (for entire disk) is entered.

7. The next prompt asks if the disk format should be verified. The verify operation checks for bad blocks and reformats those tracks that have bad blocks. This operation also attempts to map out bad blocks. The verify operation is normally chosen. The verify prompt is as follows:

```
Verify?
```

   If a Y is entered, the formatting and verifying proceed as instructed.

   If any errors occur, they are displayed. However, no errors should occur during normal system operation.

NOTE

If any character is typed during the format operation, the program is terminated. If the program is terminated in this way, no configuration record is written.

6.13.2 The CONFIGURE Program

The CONFIGURE program defines Winchester or floppy disk unit numbers or changes existing unit numbers. (A help file CONFIGHELP is available in the program. Typing an H at the top level of commands causes the help file to be displayed.) Only a few volumes on the Winchester disk should be defined initially; additional volumes can be created as needed.

| NOTE |

> The CONFIGURE program does not run on machines with only 64 kbytes of memory. In order for CONFIGURE to run successfully on a Winchester disk, WFORMAT must have placed a configuration record on the Winchester disk.

The CONFIGURE program is used to allocate unit numbers to designate either floppy drives or logical partitions on a Winchester disk. This allocation is referred to as the system configuration. The operating system uses an internal system configuration record to decide which unit number corresponds to which floppy drive or which logical partition on a Winchester disk. Each Winchester drive on the system has a configuration record that describes the physical characteristics of that disk (the number of cylinders, heads, and blocks per track) and the partitioning of the disk into different logical units.

The III.Ø Operating System builds up the system configuration record by reading the configuration record for each individual drive. The operating system combines these records into one record. To the operating system, the units that are logical partitions on a Winchester disk look like floppy disk drives. This feature allows one large Winchester disk to act like a number of smaller and more usable floppy disk drives.

The CONFIGURE program allows the user to manipulate either the system configuration record (in system mode) or a configuration record from an individual drive (in drives mode). The system configuration record is the configuration information used by the system. The program maintains temporary copies, in memory, of the configuration records from each of the drives. The system configuration record is the combination of the individual configuration records from the drives. The temporary copies of the individual configuration records are called work records.

Usually, the user works only with the system configuration record. The individual drive records are automatically created from the system configuration record when it is written out to the drives.

Only under unusual circumstances would the drive records be individually changed. For example, a floppy disk drive record might be changed so that a different allocation of unit numbers to floppy drives is used when the system is booted from a particular floppy drive. This change might be because some program that is used only occasionally should use a floppy disk instead of a Winchester unit.

The following steps explain how to run the CONFIGURE program to set up the system configuration (define blocked units). All Winchester drives to be used must have at least the minimal configuration record written by the WFORMAT program.

During configuration, some Winchester volumes should be defined to correspond to the length in blocks of a floppy disk volume. For example, if double-sided double-density floppy disks are used, a Winchester volume with 1976 blocks allows a volume-to-volume transfer between the floppy disk and the Winchester disk.

1. To execute the CONFIGURE program, an X is typed from the system command line. In response to the file name prompt, CONFIGURE is entered.

2. After a short delay, the following prompt line appears.

```
---------------------------------------------------------------------------
|Configure [system]: E(dit, S(how, R(ead, W(rite, M(ode, G(et, P(ut, H(elp ? |
---------------------------------------------------------------------------
```

If a ? is entered, the additional prompt shown below appears.

```
---------------------------------------------------------------------------
|Configure[system]:  L(ist, Z(ap, Q(uit ?                                 |
---------------------------------------------------------------------------
```

These commands are briefly described in the following paragraphs.

E(dit -- Edit a work record.

- system mode -- Usually, the system configuration record is edited because this configuration results every time the system boots from the Winchester disk.

- drives mode -- An individual drive configuration record can be edited if the configuration is to be different every time the system boots from a particular floppy drive (different floppy units) or if a conflict exists between the Winchester units on a different set of drives.

S(how -- Display either the current (the configuration being used by the operating system) or the new (the configuration record being created) system configuration that would result if the work records in memory were actually written to the drives and the system rebooted.

R(ead -- Initialize the work records in program memory, either from the current system configuration or from the configuration records on the actual drives.

|NOTE|

The program is automatically initialized from the drive configuration records when the program is executed.

W(rite -- Write the configuration record out to the drives.

- system mode -- If the system configuration record is written, all Winchester drives receive a configuration record derived from the new system configuration record.
- drives mode -- Writing an individual drive configuration record is done only when the record for the individual drive has been edited, and the new drive configuration record should actually be put on the drive.

M(ode -- Changes the mode of interaction. The modes are (1) system and (2) drives. The mode is shown in square brackets ([system] or [drives]) in the main CONFIGURE command line.

The program automatically begins in system mode. In system mode, all operations are done on a work record that reflects what the actual operating system configuration record will contain.

In drives mode, the configuration record for each individual drive can be changed. In the rare case in which a conflict exists between the unit number definitions and certain defined units (the units become inaccessible to the operating system), the drive configuration record can be changed to correct the conflict. This unusual case is only likely to happen if the Winchester drive is removed from one computer system and moved to another.

G(et -- Pulls a backup of the work version of the system configuration from either a file or the current system configuration.

P(ut -- Saves the work version of the system configuration in a file. (Serves as a backup version of the system configuration.)

H(elp -- Displays a brief description of the CONFIGURE programs and lists the various commands.

The commands listed on the second prompt line are described below.

L(ist -- Lists either the current or new system configuration (same as S(how command except for destination of information) to a file instead of the console.

Z(ap -- Removes a configuration record on a selected drive.

Q(uit -- Leaves the CONFIGURE program.

The commands described previously often have prompts or related subcommands associated with them. The following paragraphs describe the actions performed.

|NOTE|

To show a range, such as 1-123, square brackets enclose the numbers. For example, [1..123] denotes the numbers between 1 and 123.

E(dit -- The prompt line is as follows:

```
-----------------------------------------------------------------------------
|Config Edit: P(rint record, modify F(loppy or W(inchester units, Q(uit, ? |
-----------------------------------------------------------------------------
```

- Print record -- Displays the configuration record being edited.
- Modify F(loppy or W(inchester units?

  -- Entering an F causes the following additional prompt line to appear:

```
-----------------------------------------------------------------------------
| Edit Floppy set: A(dd or R(emove units, Q(uit, ?                          |
-----------------------------------------------------------------------------
```

If an A is entered, the following prompt appears:

```
-----------------------------------------------------------------------------
| Add Floppy unit [..]?                                                     |
-----------------------------------------------------------------------------
```

A unit number is entered in response to this prompt. The number entered is added as a new unit to the defined floppy units.

If an R is entered, the following prompt appears:

```
-----------------------------------------------------------------------------
| Remove Floppy unit [..]?                                                  |
-----------------------------------------------------------------------------
```

The unit number to be removed from the defined floppy unit is entered.

For both commands, A(dd and R(emove, the floppy unit set is displayed.

-- Entering a W causes the following additional prompt line to appear:

```
------------------------------------------------------------------------------
| Edit Winchester set: A(dd, C(hange or R(emove unit numbers, Q(uit, ?       |
------------------------------------------------------------------------------
```

If an A is entered, the following prompt appears:

```
| Add Winchester unit [..]?                                              |
```

After the number of the unit to be added to the defined Winchester units is entered, a prompt asking for the number of blocks in the unit appears. Once the number of blocks is entered, the unit is added.

If a C is entered, the following prompt appears:

```
| Change Winchester unit [..]?                                           |
```

If the unit number entered is a valid Winchester unit, the following prompt appears:

```
| New Winchester unit number [..]?                                       |
```

Once the new number is entered, the unit number of the existing Winchester unit is changed.

If an R is entered, the following prompt appears:

```
| Unit ___ with ___ blocks is the last unit on drive ___.                |
| Remove it?                                                             |
```

Units on the Winchester disk are removed from the back forward. Therefore, the R(emove command gives details regarding the last unit on the drive and asks for verification of removal. Once a Yes entered, the last Winchester unit defined on the specified Winchester drive is removed.

S(how — The prompt is as follows:

```
| C(urrent system, N(ew system?                                          |
```

Entering a C causes the current system configuration to be displayed.

Entering an N causes the following prompt line to appear.

```
--------------------------------------------------------------------------
|Show config of New system assuming system on F(loppy or W(inchester, Q(uit, ?|
--------------------------------------------------------------------------
```

If an F is entered, the program prompts for the floppy drive number. After the number is entered, the system configuration is displayed as if the system had booted from that floppy drive.

If an N is entered, the program assumes the new system on Winchester drive Ø. Drive Ø is assumed because the only Winchester drive that can be booted from is drive Ø. The system configuration record is displayed as if the system had booted from Winchester drive Ø.

R(ead -- The following message and prompt appear:

```
--------------------------------------------------------------------------
| Record Initialization                                                  |
|                                                                        |
| Do you want to re-initialize?                                          |
--------------------------------------------------------------------------
```

After the prompt, a warning message is displayed. If a Y is entered, the following prompt appears:

```
--------------------------------------------------------------------------
| Get configuration from D(rives or current S(ystem, Q(uit, ?            |
--------------------------------------------------------------------------
```

The work records in memory can be reinitialized to be identical with either the drive configuration records or the configuration record currently being used by the operating system.

W(rite -- The following prompt appears:

```
--------------------------------------------------------------------------
| Write out the System configuration?                                    |
--------------------------------------------------------------------------
```

If a Y is entered, the current system work record is written to the drives.

M(ode -- The following message and prompt appear.

```
--------------------------------------------------------------------------
| In the current mode you are working on the system configuration record. |
|                                                                        |
| Do you want to change the mode?                                        |
--------------------------------------------------------------------------
```

Changing the mode allows the configuration record of an individual drive to be changed independent of the system configuration.

G(et — The following prompt appears:

```
-----------------------------------------------------------------------
| Get the system configuration from what file?                        |
-----------------------------------------------------------------------
```

This command retrieves a system configuration that was previously P(ut in the specified file.

P(ut — The following prompt appears:

```
-----------------------------------------------------------------------
| Put the system configuration record in what file?                   |
-----------------------------------------------------------------------
```

This command saves a copy of the current system configuration work record in the specified file.

H(elp — This command displays the help file CONFIGHELP.

6.13.3 Using BOOTMAKE To Put an H3 Bootstrap On a Winchester Disk

If a new configuration record has been placed on a Winchester disk and no Winchester units were previously defined, the system must be rebooted so that the operating system reads the new configuration record. To put an H3 bootstrap on the Winchester disk, the BOOTMAKE program must be executed. The following steps describe the program execution.

1. An X is entered from the system command line. In answer to the prompt regarding the file to execute, BOOTMAKE is entered. Specify the new boot, and in response to the code file name prompt, enter BOOT.

2. The program prompts for the unit number on which the bootstrap is to be placed. Any unit on the Winchester disk is acceptable; however, the bootstrap is always placed on the same location on the Winchester disk regardless of the specification of a certain unit from which to boot.

   The number of the unit on which the bootstrap is to be placed is entered.

3. Next, the program asks for the name of the codefile containing the bootstrap program.

   The following response should be entered:

   BOOT <ret>

4. After a short pause while the bootstrap is processed, a prompt appears requesting the memory size of the machine on which the bootstrap is to be used.

   The following response should be entered:

   128 <ret>

   BOOTMAKE now places the bootstrap on the specified unit.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

6.14 PATCH

The PATCH utility program allows the user to examine and/or change a block or blocks on disk. This program is useful, for example, if a file is inadvertently removed from the directory but may still be written on the disk. The user can examine all portions of the disk marked as <unused> to determine if the data are recorded on the disk. Another typical use for PATCH is to update the directory (number of blocks) on a newly created disk that was made using a volume-to-volume transfer of a single-sided to double-sided diskette. The smaller number (988) can be changed in the block to the larger number (1976).

The PATCH program is executed by typing an X from the system main command line. In response to the file name prompt, PATCH is entered.

The first prompt line to appear is as follows:

```
| Patch [H3]:  F(ile, Q(uit                                                |
```

Entering a Q (for Q(uit) terminates the program.

Entering an F (for F(ile) causes the following prompt to be displayed:

```
| Filename : <cr for unit i/o>                                             |
```

The file name (including the volume prefix and file type suffix) of the file to be patched may be entered. If a (<ret>) is entered, the following prompt appears:

```
| Unit to patch?                                                           |
```

An I/O unit number can be entered in response to the prompt. (The pound sign and colon are not required.) For example, a 5 could be entered to specify unit #5. Entering a <ret> is useful if the disk (or other device) has no directory or has some problem with the directory.

After either a file name or unit number is entered, the following prompt line appears offering a choice of actions.

```
| Patch [H3]: G(et, Next, B(ack, F(ile, Q(uit                              |
```

These options are briefly described below:

- G(et - The Get option allows a specific block to be read. Typing a G causes the following prompt to appear:

```
--------------------------------------------------------------------------
| Block?                                                                 |
--------------------------------------------------------------------------
```

- N(ext - The Next option "gets" the next block after the current block (ascending sequential order). The new block number appears on the prompt line in brackets.

- B(ack - The Back option "gets" the block before the current block (descending sequential order). The new block number is displayed on the prompt line in brackets.

- F(ile - The File option allows another file to be specified as the file to be patched.

- Q(uit - The Quit option terminates the program.

Either the Get option (get a block) or the File option (specify a file to patch) must be performed first to provide a starting point.

In response to the prompt for a block number, the block to be examined of the specified file or unit is entered.

| NOTE |

No range checking is provided on this read.

If a file name is entered to be patched, the block numbers specified are file relative. That is, block 10 is block 10 of the file. If a block number greater than the number of blocks in the file is entered, the last block of the file is displayed. Likewise, if a unit is specified the block number given is unit-relative.

Once the block number is specified, the following prompt appears:

```
--------------------------------------------------------------------------
| Patch [H3]: G(et, Next, B(ack, H(ex, M(ixed, F(ile, Q(uit [nnnn]       |
--------------------------------------------------------------------------
```

The current block number appears enclosed in brackets [nnnn], and two additional options are available.

- H(ex - The Hex option causes the block to be displayed entirely in hexidecimal characters.
- M(ixed - The Mixed option causes the block to be displayed in ASCII characters where possible, and hexidecimal values elsewhere.

Once the block is displayed in response to either an H or an M, the following prompt line appears across the top of the screen to allow the block to be changed.

```
| Alter:pad vector 1,5,3,Ø Ø..F hex characters, S(tuff, Q(uit [nnnn]       |
```

The vector keys on the terminal control cursor movement. The cursor does not move off the data. Typing a hex character changes the character at the location of the cursor only if one or more of the data positions is changed.

The S(tuff command displays the following prompt line:

```
| Stuff for how many bytes:                                                |
```

A number in the range Ø to 512, followed by a <ret>, causes PATCH to accept the number. The next prompt line is as follows:

```
| Fill with what hex pair?                                                 |
```

If hexadecimal, the byte value is entered. The data reappear on the screen with the number of bytes specified filled with the value specified. Filling starts with the location of the cursor.

To return to the previous prompt line, a Q must be typed to exit the alter mode. After a change is made to the block and a Q entered to quit the block the prompt line reappears but has the additional command: P(ut. P(ut is the next command to be executed to write the changed data back from where it was read.

Figure 6-1Ø gives an example of a block examined through the PATCH program. The user commands and input are shaded. Comments are enclosed in braces ({}).

```
Command: E(dit,R(un,F(ile,C(omp,L(ink,X(ecute,D(ebug?

     X
     Execute what file? PATCH

     PATCH [H3]: F(ile, Q(uit

     F
     Filename: <cr for unit i/o) <ret>
     Unit to patch? 5<ret>
     PATCH [H3]: G(et,N(ext,B(ack,H(ex,M(ixed,F(ile,Q(uit
     G
     Block:8Ø
     PATCH [H3}: G(et,N(ext,B(ack,H(ex,M(ixed,F(ile,Q(uit [8Ø]
     M
     Alter: pad vector 1,5,3,Ø Ø..F hex characters,S(tuff,Q(uit [8Ø]

    {The display below is an approximation of the screen display of block 8Ø.}


      | Ø:   2:   4:   6:   8:   1Ø:  12:  14:  16:  18:  2Ø:  22:  24:  26:  28:
    Ø:|1ØA   3.   2     L   2    E D  I T  O R   ØD 1ØF  - -  - -  - -  - -  -ØD
   3Ø:|1ØF ØD1Ø   FØD 1Ø %  T h  e    L 2   E   d i  t o  r    i s   a    v  e r
   6Ø:| si  o n    o  f   t h  e    s c  r e  e n  - o  r i  e n  t e  d    E d
   9Ø:| it  o r    w  h i  c h   a   l l  o w  s   ØD1Ø  % e  d i  t i  n g   o
  12Ø:| f   f i   l e  s   w h  i c  h    a r  e    t o  o    l a  r g  e    t o
  15Ø:|  f  i t    i  n t  o    t h  e    m a  i n    m  e m  o r  y    b u  f f
  18Ø:| er  .ØD   1Ø %  T h  i s   e  d i  t o  r    a u  t o  m a  t i  c a  l l
  21Ø:| y   p r   o d  u c  e s   a   b  a c  k u  p    c o  p y   o   f    t h
  24Ø:| e   f i   l e   b   e i  n g  ØD 1Ø %  e d  i t  e d  .     B   e c  a u
  27Ø:| s e   t   h e   L   2    E d  i t  o r   i   s    a n   e   x t  e n  d e
  3ØØ:| d   v e   r s  i o  n    o f   t  h e   S   c r  e e  n -  ØD 1Ø %  O r
  33Ø:| ie  n t   e d   E   d i  t o  r ,   v   e r  y    f e  w    d i  f f  e r
  36Ø:| en  c e   s    e x  i s  t    b e  t w  e e  n    t h  e    t w  o    e d
  39Ø:| it  o r   s .           ØD 1Ø %  T h  e s  e    d i  f f  e r  e n  c e
  42Ø:| s   a r   e    d e  s c  r i  b e  d    i n   t  h e   f  o l  l o  w i
  45Ø:| ng   s    u b  s e  c    i o  n s  .ØD 1Ø % ØD1Ø  %ØD 1Ø %  3 .  2 .  1
  48Ø:|  I  n i   t i  a t  i n  g    t h  e    L 2   E   d i  t o  rØD 1Ø ,  - -
  512:| —|


     Q
     PATCH [H3]: G(et,N(ext,B(ack,H(ex,M(ixed,F(ile,Q(uit [8Ø]
```

Figure 6-1Ø. Using PATCH to Examine a Block.

6.15 DEBUGGER

The DEBUGGER may be used to debug user programs running on the operating system or to debug the operating system itself. DEBUGGER, when invoked, may insert or delete breakpoints in the work file or may break at breakpoints in the work file.

Use of DEBUGGER requires familiarity with the UCSD III.Ø Operating System and the Compiler. Often a compiler-generated listing of the program being debugged is necessary. Additionally, a disassembly listing may be required so that breakpoints can be inserted with reference to segment number, procedure number, and offset within a procedure.

DEBUGGER is divided into two major components: (1) the Breakpoint Handler and (2) the Debugger. The Breakpoint Handler is invoked from the system main command line. The Debugger is invoked when a breakpoint is encountered in an executing work file and/or when a run-time error occurs in any program. The Breakpoint Handler and the two invocations of the Debugger are described in the following subsections.

6.15.1 The Breakpoint Handler

To invoke the Breakpoint Handler, enter D (for D(ebug) from the system main command line:

```
| Command: E(dit, R(un, F(ile, C(omp, L(ink, X(ecute, D(ebug?    (1)   |
```

If a code file does not currently exist, the system compiles the work file (as if R(un had been entered).

The following Breakpoint Handler prompt then appears:

```
| Debug: R(esume, I(nsert, L(ist, C(lear breakpoints, Q(uit ?    (2)   |
```

The following list of commands explains the Breakpoint Handler options and shows further prompts as options are selected.

R(esume — Continues running the user program.

I(nsert — Inserts one or more breakpoints (the maximum number of breakpoints that can be inserted is 1Ø; that is, max number = 1Ø). For each breakpoint, the Breakpoint handler prompts:

```
----------------------------------------------------------------------
|         Enter segment number:  (enter number in decimal);     ③    |
|         Enter procedure number:           -----                    |
|         Enter procedure IPC:              -----                    |
----------------------------------------------------------------------
```

Validity checking is done by the Debugger for each value to assure that a breakpoint is placed over a P-code operator. This checking is done because a breakpoint is a P-code operator (RPU) and must replace an operator not an operand. If the insertion is successful, information about the breakpoint is displayed. The information displayed is shown below.

```
----------------------------------------------------------------------
|Index:   S# <seg> P# <proc> IPC <proc-ipc>(in hex)  Op-code <op>( in hex)|
----------------------------------------------------------------------
```

After the information about one breakpoint is displayed, the Breakpoint Handler prompts:

```
----------------------------------------------------------------------
|Insert another breakpoint ? (Y or N)                                |
----------------------------------------------------------------------
```

A "Y" reply goes back to ③, and a "N" reply stops the insertion and goes back to ② .

L(ist — Lists all breakpoints or alternatively, displays "No breakpoints", which then returns to ② , the main Breakpoint Handler command line.

C(lear — Clears breakpoints. The Breakpoint Handler prompts:

```
----------------------------------------------------------------------
|A(ll, S(ingle ?                                                     |
----------------------------------------------------------------------
```

An "A" (for A(ll) reply clears all breakpoints and displays the same information as for an inserted breakpoint for each breakpoint removed.

An "S" (for S(ingle) reply clears only a single breakpoint. The Debugger, however, lists all breakpoints then prompts:

```
------------------------------------------------------------------------
|Clear breakpoint with index = (enter selected index   (4)            |
|                                number, as listed)                   |
------------------------------------------------------------------------
```

After the number is entered if the clearing is successful, the Breakpoint Handler prompts:

```
------------------------------------------------------------------------
|Continue clearing ? (Y or N)                                         |
------------------------------------------------------------------------
```

A "Y" reply takes the user back to (4) . An "N" reply takes the user back to (2) .

Q(uit Returns control to the operating system (takes user back to (1) the Outer Level command line).

The breakpoint information is kept in block zero of the code file. The layout of block zero is shown below:

```
    blockØlayout= record
                     otherdata: array [Ø..224] of integer;
                     bkcntrl   : packed record
                                   bkcnt: Ø..maxbrk  {breakpoint count}
                                 end;
                     bkinfar   : array [Ø..maxindx]  of {bp info array}
                                   packed record
                                     relsblk,saveop,opseg,opproc: byte;
                                     opsipc: integer;
                                     oppipc: integer;
                                   end;
                  end;
 where
    maxbrk {max. number of breakpoint} = 1Ø;
    maxindx {max. index value} = maxbrk-1= 9.
```

6.15.2 The Debugger

The Debugger is either (1) called when a breakpoint is executed or (2) called when a run-time error occurs.

DEBUGGER (Breakpoint Call)

When a breakpoint is executed, DEBUGGER is invoked, and the following message and information are displayed:

```
|Programmed break-point                                         |
|S# <seg.number>, P# <proc.number>, I# <proc-ipc>               |
```

Then, the Debugger portion of DEBUGGER is invoked, the Status (see S(tatus command option) is displayed, and the Debugger command prompt is shown.

```
|Debugger: R(esume, D(ump, B(reakpoint, X(amine, S(tatus, Q(uit ?   5 |
```

The following list discusses each of the Debugger command options.

R(esume — Continues running the work file.

D(ump — Dumps the entire memory into a user specified file that may be placed on any volume. If the dump is to disk and insufficient room exists, the Debugger creates a partial memory dump if user requested. A memory dump from addresses Ø-7FFF is performed on a machine with 64K bytes of memory. For a machine with 128K bytes of memory, addresses Ø-FBFF are dumped. The Debugger prompts as below:

```
|Input your Notice:  (max size= 8Ø char.)                        |
```

This notice (or <space>) is saved in block Ø of the dump file along with the following information:

- The contents of registers -3..13.
- The run-time error code that caused Debugger invocation.
- The segment#, proc#, and the IPC (Instruction Program Counter) of the corresponding opcode.
- The date (as displayed at boot time).

The record describing this dumped information is as follows:

```
dumplayout = record
                 regs: array[Ø..16] of integer;
                 errcode: integer;
                 seg: integer;
                 proc: integer;
                 ipc: integer;
                 date: daterec; { 1 word }
                 filler: array[Ø..193] of integer;
                 notice: packed array[Ø..79] of char;
             end;
```

The memory output may be viewed as decimal, hex, binary, ASCII, or unsigned decimal. Also, the memory dump can be stopped by typing any character.

B(reakpoint — Control goes to the Breakpoint Handler. Very much like I(nsert of Breakpoint Handler except:

- The code file in memory is updated correspondingly for I(nsert and C(lear.
- Q(uit returns to 5, instead of 1.

X(amine — Goes to the memory X(amine mode. The following command prompt line appears:

```
| C(hain, O(ffset, M(emory, re-D(isplay, A(lter,                     |
| S(tatus, R(adix, T(asks, output F(ile, Q(uit?                      |
```

The following list gives a summary of these commands.

C(hain — Moves the addressing environment pointer (current MP) along dynamic or static links. The following prompt appears:

```
|S(tatic, D(ynamic  ?                                                |
```

If D (for D(ynamic) is entered, the MP pointer follows the dynamic link chain field in the mark stack control word.

If S (for S(tatic) is the reply, the MP pointer follows the static link chain of the mark stack control word.

The Debugger then displays either the static or dynamic chain of mark stack control words. An example display for dynamic chaining is as follows:

```
Dynamic Calling Chain for This Task
-----------------------------------

1) S# 8, P# 68, I# 1Ø
2) S# 8, P# 1, I# 68
3) S# 1, P# 1, I# 11
```

The display shows the chain of mark stack control words displaying the segment number, procedure number, and IPC. The Debugger requests the mark stack number to change context. Entering a Ø keeps the current mark stack context.

O(ffset — Displays the contents of memory at a word offset from the current MP (see C(hain). Offset allows access of values of variables because variables are allocated at offsets from a mark stack. The offset corresponds to the variable offsets assigned by the Compiler.

This command displays the following prompts (the Debugger prompts if the input data are required in hex):

```
------------------------------------------------------------------------
|Offset=  { enter the offset value }                                   |
|Length=  { enter number of WORDS to be displayed }                    |
------------------------------------------------------------------------
```

The requested words are then displayed as below:

```
<base address> offset : 1
   <val>,<val>, ...... for the length requested
```

re-D(isplay — Displays the previous O(ffset or M(emory just displayed in hex, decimal, ASCII, binary, or unsigned decimal. This mode is used for future outputs of M(emory or O(ffset until either R(adix or other re-D(isplayed are entered.

A(lter — Modifies one word in memory. The Debugger displays the current radix and prompts:

```
--------------------------------------------------------------------------
|Enter address: ( in current radix )                                      |
--------------------------------------------------------------------------
```

After the address is entered, the Debugger displays the following prompt and then asks how many words at the starting address should have the new value.

```
--------------------------------------------------------------------------
|was: xxxx   Change to: zzzz                                              |
--------------------------------------------------------------------------
```

M(emory — Displays the contents of memory from the start address to the end address. Addresses are in words. The Debugger memory prompts are below.

```
--------------------------------------------------------------------------
|Start address =                                                          |
|End address   =                                                          |
--------------------------------------------------------------------------
```

S(tatus — Displays the environment status. (See discussion of S(tatus as described for the Debugger 5.)

R(adix — Switches the current radix mode from decimal (the default) to hexadecimal or vice versa. The following prompt is displayed:

```
--------------------------------------------------------------------------
|Radix switched from Decimal to Hex (or Hex to Decimal).                  |
--------------------------------------------------------------------------
```

| NOTE |

This option is always reset to decimal when the Debugger is first invoked (because of a run-time error).

T(asks — Displays the active user tasks. An example display is shown below:

Currently Active Tasks

Main task: 1) S# 8, P# 68, I# 1Ø

The Debugger then requests the task number to change context. With a new context selected, c(haining may be done on the new task. Entering a zero keeps the context at the present task where the segment number, procedure number, and IPC of each task are displayed.

output F(ile — Allows Debugger output to be directed to a file other than the CONSOLE:.

R(esume — Continues running the work file.

Q(uit — Goes back to the Debugger command line 5 .

| NOTE |

Because of the mechanism used to return from a breakpoint, the active breakpoint in memory is replaced by the original P-code and is restored only when another breakpoint is encountered. Therefore, a single breakpoint is NOT restored until another one is encountered; however, it is still preserved in the code file.

S(tatus — Displays the environment status. The user program MP is the pointer to the MSCW (mark stack control word) of the user program at the time the breakpoint occurred.

The current MP is the pointer to the current activiation record as performed by the C(hain command. (See C(hain command.)

| NOTE |

At the first call of the Debugger, user program MP=current MP.

The display of the current status is in the form shown below. The addresses below are shown in current Radix.

```
------------------------------------------------------------------
|                Address of tib :      0040                      |
|                    ready queue            :     0A22            |
|                    system segment vector  :     007F            |
|                    priority               :     007F            |
|                    splow                  :     632E            |
|                    spupr                  :     D3C0            |
|                    sp                     :     A514            |
|                    mp                     :     A514            |
|                    bp                     :     D39E            |
|                    ipc                    :     01C3            |
|                    segbase                :     E50F            |
|                    hangp                  :     A51E            |
|                    ioresult               :     0000            |
|                    segment vector         :      NIL            |
|                    maintask               :     TRUE            |
|                    startmscw              :     D3CE            |
|                                                                 |
|                    User mp                :     A55F [8,68]     |
|                    User bp                :     D39E [1,1]      |
|                    Current mp             :     A55F [8,68]     |
------------------------------------------------------------------
```

Q)uit Returns to 1 , the Outer Command level.

DEBUGGER (Run-Time Error Invocation)

The Debugger is called by any run-time error other than a stack overflow error. The following prompt appears:

```
-----------------------------------------------------------------------
|D(ebug or Type <space> to continue                                   |
-----------------------------------------------------------------------
```

If the response is to type <space>, the usual path of execution for an error is followed. That is, the system reinitializes itself.

If D (for D(ebug) is entered, the Debugger is called, and the Debugger command line 5 appears.

If adequate space is not available to load the Debugger (about 60 words in the stack space and 6000 words on the heap), the system responds:

```
-----------------------------------------------------------------------
|Not enough room for Debugger                                         |
-----------------------------------------------------------------------
```

If the Debugger is invoked as a result of a run-time error, the invocation of Breakpoint Handler is not allowed because a program other than the work file may have been executed from the Outer Level commands.

6.15.3 Accessing User Program Variables From the Debugger

In order to access variables declared in a user program, a compiler produced listing is required. Refer to Figure 6-11 for a compiler produced listing of an example program. Refer to Section 5.4.3 for a description of a compiler produced listing.

```
 1  128   1:D     1 {$L+}
 2  128   1:D     1 program test;
 3  128   1:D     1 var i: integer;
 4  128   1:D     2     ch: char;
 5  128   1:D     3     s: string[3];
 6  128   1:D     5     k: integer;
 7  128   1:D     6
 8  128   2:D     1   procedure p;
 9  128   2:D     1   var j: integer;
10  128   2:D     2       chl: char;
11  128   2:D     3       ll: integer;
12  128   2:0     0   begin
13  128   2:1     0     j := 21;
14  128   2:1     3     chl := 'I';
15  128   2:1     7     ll := 55;
16  128   2:0    11   end;
17  128   2:0    14
18  128   1:0     0 begin
19  128   1:1     0   i := 0;
20  128   1:1     7   ch := 'A';
21  128   1:1    11   s := 'xyz';
22  128   1:1    19   k := 44;
23  128   1:1    23   p;
24  128   1:0    25 end.
```

Figure 6-11. Compiler Produced Listing.

In this program assume the I(nsert command in the Breakhandler has been used to insert a breakpoint in segment 128, procedure 2, IPC 11, which is after the last statement in procedure P. When this breakpoint is encountered during execution of the program, the program stops, and the Debugger displays the following:

```
-----------------------------------------------------------------
|          Programmed break-point                               |
|          S# 128, P# 2, I# 12                                  |
|          Deb                                                  |
|          Op-code: 150 (96)                                    |
|          Debugger: R(esume, D(ump, B(reakpoint, X(amine, S(tatus, Q(uit |
-----------------------------------------------------------------
```

If an X for X(amine is typed, the Debugger displays the following prompt:

```
-----------------------------------------------------------------
|C(hain, O(ffset, M(emory, re-D(isplay, A(lter                  |
|S(tatus, R(adix, T(asks, output F(ile, Q(uit                   |
-----------------------------------------------------------------
```

The C(hain and O(ffset commands are used to access user variables. The C(hain command moves through addressing environments established by procedure calls. The O(ffset command accesses variables as determined by the rightmost numbers in the declaration parts of the compiler produced listing. In the above example, in the procedure Paddressing environment for offset 1 and length 3, the Debugger displays:

```
-11436 offset:  1
    21    73   55
```

The value -11436 is the two's complement representation of the memory address where the program variables in this addressing environment start. Variable J is at offset 1; CH1 is at offset 2; and LL is at offset 3.

The C(hain command can be used to change the addressing environment to the outer block of the example program. To make this change, type C(hain; the Debugger then displays the following prompt line.

```
-----------------------------------------------------------------
| S(tatic, D(ynamic ?                                           |
-----------------------------------------------------------------
```

If D(ynamic is entered, the Debugger prompts:

```
-----------------------------------------------------------------
|DYNAMIC CALLING CHAIN FOR THIS TASK                            |
|-----------------------------------                            |
|  1)  S# 128, P# 2, I# 12                                      |
|  2)  S# 128, P# 1, I# 25                                      |
|Enter number beside procedure to look at (0 to leave as is):   |
-----------------------------------------------------------------
```

If a 2 is entered, the environment is moved to the outer block. Typing O(ffset allows the values of the variables in the outer block to be examined, as shown below:

```
-------------------------------------------------------------------------
|  Offset = 1                                                           |
|  Length = 5                                                           |
|  -11429 offset:   1                                                   |
|       Ø   65   3Ø723    31353     44                                  |
-------------------------------------------------------------------------
```

If re-D(isplay is entered, the following prompt appears:

```
-------------------------------------------------------------------------
| mode:                                                                 |
-------------------------------------------------------------------------
```

In response to the "mode:" prompt, enter A(SCII. The following display appears:

```
-------------------------------------------------------------------------
| ^@   ^@   A^@   ^@x   yz   ,   ^@                                     |
-------------------------------------------------------------------------
```

These values correspond to the variables declared in the outer block of the program.

|NOTE|

When Pascal program variables are declared in a rated by commas (for example, VAR I, J, K, L: INTEGER), the offsets are assigned in reverse order. That is, variable L is at offset 1; K is at offset 2; J is at offset 3; and I is at offset 4.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

6.16 COPY

The COPY utility program runs as a low priority background task and allows file transfers between any source and destination files while the system is used in a normal manner for other work. A common use for the COPY program is, for example, to list files to a printer while using the system for other work.

To execute COPY, an X is entered from the system main command line; the file name COPY is entered in response to the file name prompt.

COPY signals a copier task semaphore which starts the copier task which prompts for the source and destination file names for the file transfer. If a null string is entered either for the source or destination, the background copy aborts.

The COPY program may also be used to terminate a copy in the middle of its performance. That is, if the COPY program is executed while the copier task is running, the COPY program displays the following:

```
---------------------------------------------------------------------------
|Copier Busy Type <space> to continue, K(ill to terminate                 |
---------------------------------------------------------------------------
```

Entering a space causes the copier task to continue; entering a K causes the copier task to terminate.

The following restrictions apply to the use of COPY:

- The source and destination copier task files should not be removed when the copier task is active. Unpredictable results occur.

- The Filer cannot crunch the volume when the copier task is active on the volume to be crunched.

- If a run-time error occurs during copier task activity, the copier task continues to be active. However, if the system is reinitialized (by pressing Reset), the copier task terminates.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

7. PASCAL PROGRAMMING CONSIDERATIONS

Many aspects of the Western Digital UCSD Pascal III.Ø Operating System influence how a program should be written to run most efficiently on a 16ØØ Series SuperMicro Computer System. These aspects are described in this chapter of the manual.

This chapter describes the following topics:

- Introduction to the III.Ø Operating System
- Intrinsics
- Segments
- Linkages
- System Library
- UCSD Pascal Enhancements

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

7.1 INTRODUCTION TO THE III.Ø OPERATING SYSTEM

The H3 III.Ø Operating System includes automatic boot of either single- or double-density diskettes. The SB16ØØ systems automatically try both densities when booted with a diskette - regardless of the switch setting. The ME16ØØ systems also provide the automatic density feature if the floppy disk controller board is level B8 or later. In addition, diskettes with different densities may be freely exchanged during normal operation.

This operating system also supports Winchester disk drives (1Ø and 4ØMB) on the ME16ØØ series computer systems.

The H3 III.Ø Operating System offers protection during diskette removal so that the correct diskette is on line during loading of overlayable code segments. If the boot diskette is not on line when required, the operating system prompts as below:

```
| Insert OSH3 in drive 4. Type <space> when ready or <esc> to abort.   |
```

If the diskette with a user program is not on line, the operating system prompts as below:

```
|Insert PROGRAM diskette in drive _. Type <space> when ready or <esc> to abort|
```

The correct drive number is displayed in the prompt. Typing a space with the correct diskette on line causes the required code segment to be loaded. For example, removing the system diskette during editing does not cause the system to hang.

The following sections discuss aspects of software control of the III.Ø Operating System. Topics covered include code file representation, program execution, operating system structure, bootstrapping of the operating system, concurrent task (process) representation, and task control primitives.

7.1.1 Operating System Structure

The Pascal Compiler emits code that runs directly on the WD9000 microprocessor. The Compiler, Screen Editor, Operating System, and all utilities are written in Pascal and use this instruction set.

Figure 7-1 is a skeleton version of the III.0 Operating System, a large Pascal program. This subsection describes the structure and component parts of the program on the UCSD Pascal system.

```
program pascalsystem;

var syscom : syscomrec;
    ch : char;
  segment procedure userprog; forward;
  segment procedure syscode; forward;
  segment procedure cspcode; forward;

  segment procedure userprog;

      begin
        .
        .
        .
      end;

  segment procedure syscode;

      segment procedure printerror;
          begin
            .
            .
            .
          end;
```

Figure 7-1. Structure of III.0 Operating System. (Continued on next page)

```
segment procedure initialize;
    begin
      .
      .
      .
    end;

segment procedure getcmd;
    begin
      repeat
        case ch of
          'e':  editor;
          'f':  filer;
          'l':  linker;
          'x':  execute;
          'c':  compiler;
          . . .
        end { case }
      until false;
    end;
  begin { syscode }
    initialize;
    getcmd;
  end   { syscode };

segment procedure cspcode;
    begin
      ioinit;
      syscode;
    end;

begin (* pascal system *)
  cspcode;
end.
```

Figure 7-1. Structure of the III.Ø Operating System. (Continuation)

If this skeleton version were expanded to the complete Pascal system, it would consist of several thousand lines of Pascal and compile to more than 4Ø,ØØØ bytes of code.

The III.Ø Operating System consists of a code file containing several segments and operating system tables. (Segments are discussed in a following subsection.) Some segments of the operating system are always resident in main memory; other segments are resident only when necessary and are overlaid by other code when not necessary.

The segments of the III.Ø Operating System that are always resident are: segment Ø, PASCALSYSTEM; segment 2, SYSCODE; and segment 3, CSPCODE. When a user program executes, only segments Ø, 2, and 3 are resident, which leaves approximately 21,ØØØ words of memory available on a 64K byte system. Segments 4, PRINTERROR; 5, INITIALIZE; and 6, GETCMD are overlaid as necessary. The III.Ø Operating System also uses segments 16 to 3Ø for such things as the Debugger, Winchester I/O drivers, and floppy disk I/O driver.

The Compiler, Screen-Oriented Editor, and Filer are large programs that have their own code segments but that are called by the operating system. When a program executes, memory usage consists of in-core code segments of the program plus the resident code segments and system tables of the operating system. When the Compiler is loaded into memory in nonswapping mode, approximately 15ØØ words are available for use as symbol table space. In swapping mode, this figure increases to 36ØØ words. When the Editor is loaded in memory, approximately 77ØØ words are available for text file editing. When the Filer is loaded in memory, about 63ØØ words are available as buffer space. These numbers assume a 64K byte system. For a 128K byte system, approximately 32ØØØ words should be added.

Segments

Because UCSD Pascal has been extended so that a programmer can explicitly partition a program into segments, only some segments need be resident in main memory at a time. The syntax of this extension is shown in Figure 7-2. (Any syntactic objects not explicitly defined in Figure 7-2 retain their standard interpretation as defined by Jensen & Wirth: Pascal User Manual and Report.)

<program> ::= <program heading> <segment block> .

<segment block> ::= <label declaration part>
<constant declaration part> <type definition part>
<variable declaration part> <segment declaration part>
<segment body>

<segment declaration part> ::= {<segment declaration>}

<segment declaration part> ::= SEGMENT <procedure heading>
<segment block>; | SEGMENT <function heading>
<segment block>;

<segment body> ::= <procedure and function declaration part>
<statement part>

Figure 7-2. Segment Declaration Syntax.

Segment declaration syntax (Figure 7-2) requires that all nested segments be declared before the ordinary procedures or functions of the segment body. Thus, a code segment can be completely generated before processing of code begins for the next segment. This sequence is not a functional limitation, because forward declarations can be used to allow nested segments to reference procedures in an outer segment body. Similarly, segment procedures and functions can themselves be declared forward.

Segmenting a program does not change its meaning in any fundamental sense. When a segment is called, the operating system checks to see if it is present in memory because of a previous invocation. If the segment is resident, control is transferred and execution proceeds; if not, the appropriate code segment must be loaded from disk before the transfer of control takes place. When no more active invocations of the segment exist, its code is removed from memory. Clearly, a program should be segmented in such a way that (nonrecursive) segment calls are infrequent; otherwise, much time could be lost in unproductive thrashing (particularly on a system with low performance disk).

Segment Dictionary

The code file of the III.Ø Operating System is a sequence of code segments preceded by a segment dictionary. Code segments consist of a sequence of blocks, a 512-byte disk allocation quantum, and each code segment begins on a block boundary. The ordering (from low address to high address) is determined by the order that one encounters segment procedure bodies in passing through the operating system program.

The segment dictionary in the first block of a code file contains an entry for each code segment in the file. The entry includes the disk location and size (in words) for the segment. The disk location is given as relative to the beginning of the segment dictionary (which is also the beginning of the code file) and is given in number of blocks. This information is kept in the segment vector during the execution of the code file and is used in the loading of nonpresent segments when they are needed. Figure 7-3 details the layout of the table and shows representative contents for the Pascal system code file.

location	31	PASCALSYSTEM
size	9	
	0	USERPROG
	0	
	11	SYSCODE
	5864	
	24	CSPCODE
	3069	
	1	PRINTERROR
	864	
	5	INITIALIZE
	4542	
	9	GETCMD
	1379	

Figure 7-3. The Segment Dictionary.

Code Segment

A code segment contains the code for the body of each of its procedures, including the segment procedure itself. Figure 7-4 is a detailed diagram of a code segment. Each procedure of a code segment is assigned a procedure number, starting at 1 for the segment procedure, and ranging as high as 255.

All references to a procedure are made by its number. Translation from procedure number to location in the code segment is accomplished with the procedure dictionary at the end of the segment. This dictionary is an array indexed by the procedure number. Each array element is a segment base pointer to the code for the corresponding procedure. Because zero is not a valid procedure number, the zero entry of the dictionary is used to store the segment number (even byte) and number of procedures (odd byte). The outer block code is generated and appears last.

```
                        high addresses
                  odd                      even
           ____________________________________________
          | Number of procedures  | Segment Number     |
          |     in dictionary     |                    |
          |--------------------------------------------|
          | Procedure #1                               |--|
          | - - - - - - - - - - - - - - - - - - - - -  |  |
   |------| Procedure #2                               |  |
   |      | - - - - - - - - - - rest of - - - - - - -  |  |
   |  |---| - - - - - - -procedure dictionary - - - -  |  |
   |  |   |--------------------------------------------|  |
   |  |   |                                            |  |
   |  |   |              outer block code              |<-|
   |  |   |                                            |
   |  |   |--------------------------------------------|
   |  |   |code for other procedures of the Pascal system|
   |  |   |--------------------------------------------|
   |  |-->| Procedure #3                        code   |
   |      |--------------------------------------------|
   |----->| Procedure #2                        code   |
          |--------------------------------------------|
          |         Number of words in segment         |
          |____________________________________________|

                         low addresses
```

Figure 7-4. A Code Segment.

A more detailed diagram of a single procedure code section is seen in Figure 7-5. It consists of two parts: the procedure code itself and a table of attributes of the procedure. These attributes are as follows:

CONSTANT POOL: The compiler allocates the constants for each procedure here.

EXIT IC: This is a segment-base-relative byte pointer to the beginning of the block of procedure instructions which must be executed to terminate the procedure properly.

DATA SEGMENT SIZE: The data size is the size of the procedure data space (parameters and local variables) in words, excluding the markstack size.

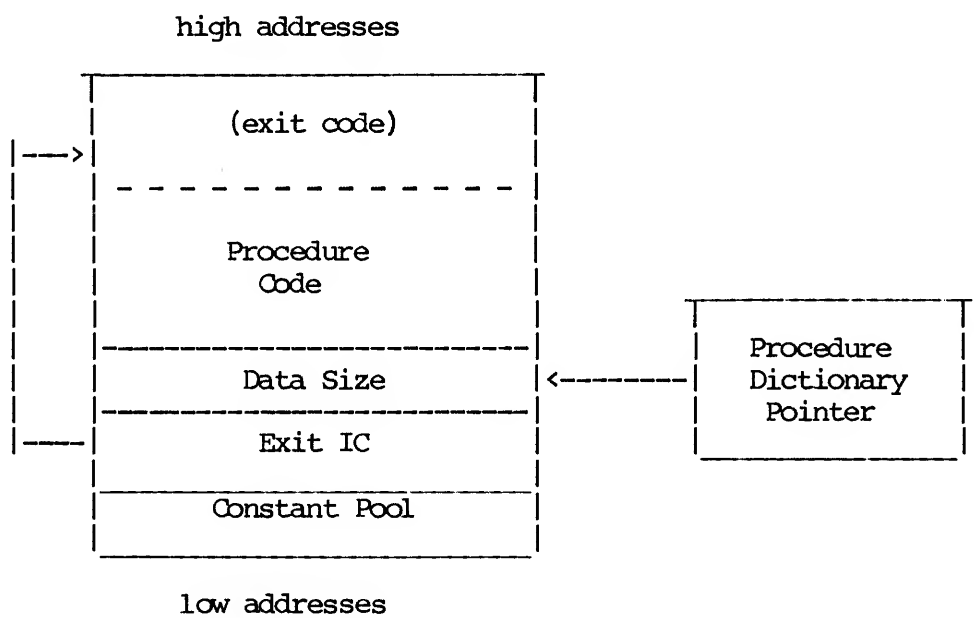


Figure 7-5. Procedure Code Section.

Figure 7-6 is a snapshot of system memory during the execution of a call to procedure GETCMD, which is the command processor of the operating system. SYSCOM serves as a communications area between the bootstrap and the operating system. The operating system tables consist of the TIB (task information block) and the segment vector, which is an array of information about active program segments. The Pascal heap is next in the memory layout; it grows toward high memory. The single stack growing down from high memory is used for 3 types of items: 1) temporary storage needed during expression evaluation; 2) a data segment containing local variables and parameters for each procedure activation; and 3) a code segment for each active segment procedure.

```
              high addresses
         _________________________
        | Code Segment Ø          |
        | - - - - - - - - - - - - |
        | Code Segment 2          |
        |    CSPCODE              |
        | - - - - - - - - - - - - |
        | markstack               |
        | - - - - - - - - - - - - |
        | Code Segment 3          |
        |    SYSCODE              |
        | - - - - - - - - - - - - |
        | markstack               |
        | - - - - - - - - - - - - |
        | Code Segment 6          |
        |    GETCMD               |
        | - - - - - - - - - - - - |
        | markstack               |
        | - - - - - - - - - - - - |
        |  <Available Memory>     |
        | - - - - - - - - - - - - |
        |                         |
        |        HEAP             |
        |                         |
        | - - - - - - - - - - - - |
        |Operating System Tables  |
        |         and             |
        |        SYSCOM           |
        | - - - - - - - - - - - - |
        | Interrupt Vectors       |
        | - - - - - - - - - - - - |
        |_________________________|
               low addresses
```

Figure 7-6. System Memory During Operating System Execution.

Consider the status of operations just before a procedure call. Conceptually, five registers point to locations in memory:

- STACK POINTER(SP) — Points to the current top of the stack.
- MARK STACK POINTER(MP) — Points to the "topmost" mark-stack in the stack, (remember that the stack grows down).
- SEGMENT POINTER(SEGB) — Points to the base of the code for the currently active segment procedure.
- INSTRUCTION PROGRAM COUNTER(IPC) — Contains the byte offset from the base of the code segment of the next instruction to be executed.
- (SPLOW) — Points to the current top of the heap and also serves as the stack limit pointer.

When a segment procedure is called, its code segment is loaded on the stack. The data segment is built on top of the stack. Figure 7-7 is a diagram of a data segment.

```
                      high addresses
                   ________________________  <--|
                   |                      |     |
                   |----------------------|     |
                   |                      |     |
                   |----------------------|     |--> local variables
                   |                      |     |
                   |----------------------|     |
                   |                      |     |
                   |----------------------|<--|
                   |  MSFLAG  |  MSSEG    |     |
                   |- - - - - - - - - - - |     |
                   |        MSIPC         |     |
                   |- - - - - - - - - - - |     |--> markstack
                   |        MSDYNL        |     |
                   |- - - - - - - - - - - |     |
      |----|       |        MSSTAT        |     |
      | MP |-->    |- - - - - - - - - - - |<--|
      |____|              low addresses
```

Figure 7-7. A Data Segment.

In the upper portion of the data segment, space is allocated for variables local to the new procedure.

In the lower portion of the data segment is a "markstack". When a call to any procedure is made, the current values of the pseudovariables, which characterize the operating environment of the calling procedure, are stored in the markstack of the called procedure. This action is so that the execution state may be restored to precall conditions when control is returned to the calling procedure.

For example, a call causes conditions in the calling procedure before the call to be stored in the markstack in the following manner:

```
MarkStack DyNamic link (MSDYNL)  <-- MP
  "    "  IPC(MSIPC)  <-- IPC
  "    "  SEGment Pointer(MSSEG)  <-- SEGB
```

The Pascal declaration for a "markstack" is:

```
TYPE    MSCW = PACKED RECORD  { MARK STACK CONTROL }
                 MSSTAT: MSCWP;          { LEXICAL PARENT POINTER }
                 MSDYNL: MSCWP           { PTR TO CALLER'S MSCW }
                 MSIPC: INTEGER;         { BYTE INX IN RETRN CODE SEG }
                 MSSEG: BYTE;            { SEG # OF CALLER CODE }
                 MSFLAG: BYTE            { CURRENTLY UNUSED }
               END  {MSCW };
```

In addition, a Static Link field becomes a pointer to the data segment of the lexical parent of the called procedure. In particular, it points to the Static Link field of the markstack of the parent. After the building of the data segment, new values for IPC, SEGB, SP, and MP are established.

7.1.2 The Bootstrap Sequence

The bootstrap sequence is initiated whenever the RESET button is pushed. Figure 7-8 (2 pages) is a flowchart describing the microcode/software instructions that are executed in order to load in and start the execution of the operating system.

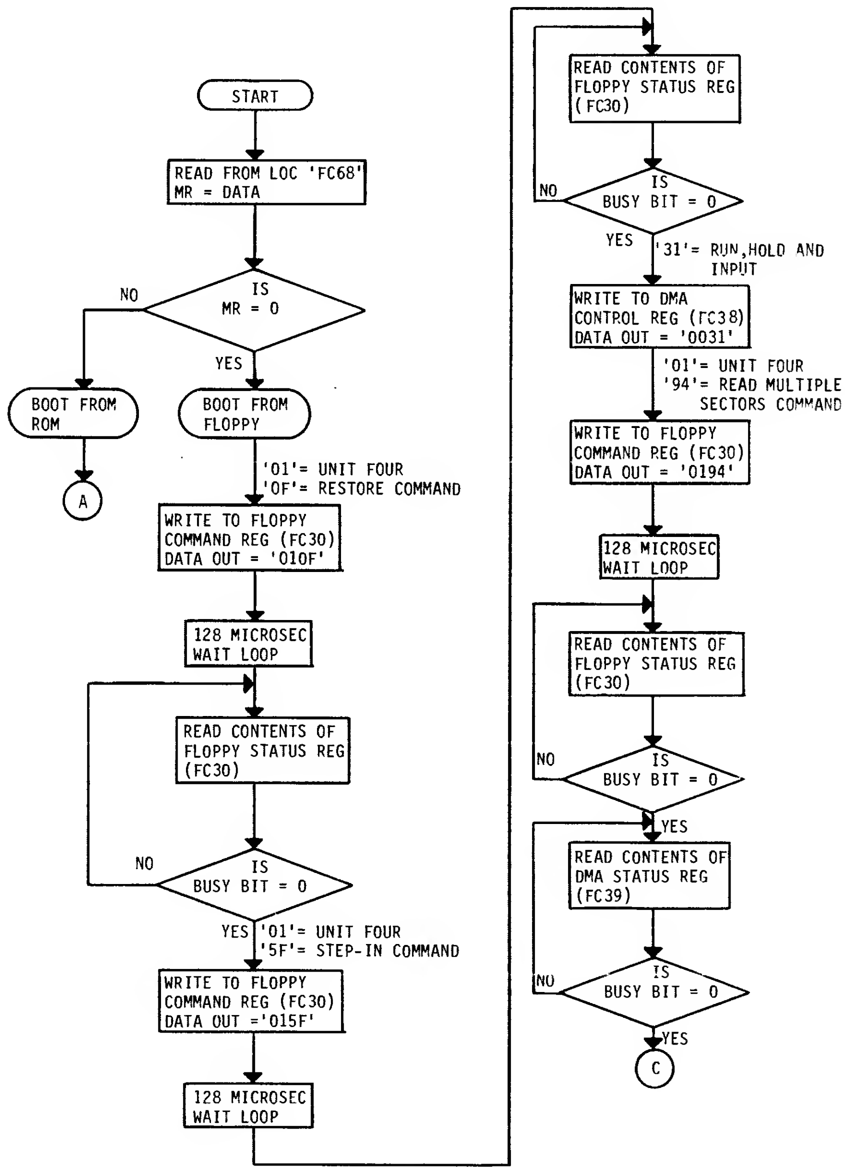
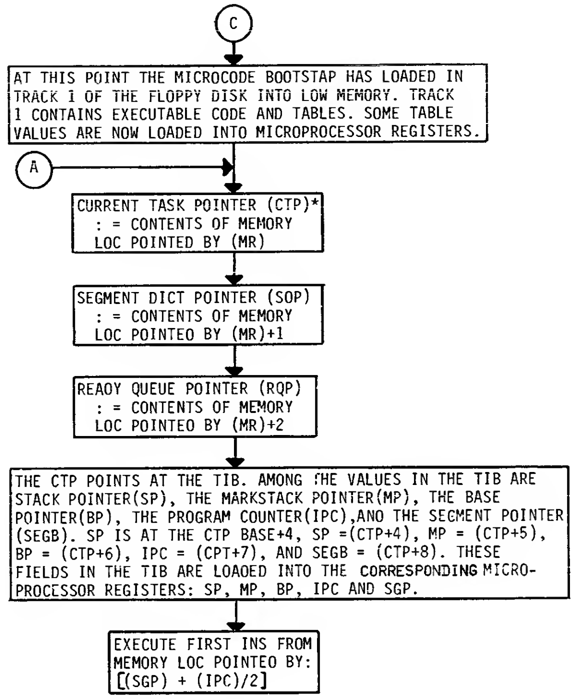


Figure 7-8. Bootstrap Microcode/Software Instruction. (1 of 2)



*THE CURRENT TASK POINTER POINTS AT THE TASK INFORMATION BLOCK (TIB). THE LAYOUT OF THE TIB IS AS FOLLOWS:

```
TIB = RECORD (TASK INFORMATION BLOCK)
      REGS: PACKED RECORD
         WAITQ: TIBP; (QUEUE LINK FOR SEMAPHORES)
         PRIOR: BYTE; (TASK'S CPU PRIORITY)
         FLAGS: BYTE; (STATE FLAGS...NOT DEFINEO YET)
         SPLOW: INTEGERP; (LOWER STACK POINTER LIMIT)
         SPURP: INTEGERP; (UPPER LIMIT ON STACK)
         SP: INTEGERP; (ACTUAL TOP-OF-STACK POINTER)
         MP: MSCWP; (ACTIVE PROCEOURE MSCW PTR)
         BP: MSCWP; (BASE AODRESSING ENVIRONMENT PTR)
         IPC: INTEGER; (BYTE PTR IN CURRENT COOE SEG)
         SEGB: ^ CODESEG; (PTR TO SEG CURRENTLY RUNNING)
         HANGP: SEMP; (WHICH TASK IS WAITING ON)
         XXX: INTEGER; (NOT USED)
         IORESULT: IORSLTWO; (RESULT OF LAST I/O OPERATION)
         SIBS: ^ SIBVEC; (ARRAY OF SIBS FOR 128..255)
         NEXTTIB: TIBP
       ENO (REGS);
     MAINTASK: BOOLEAN;
     STARTMSCW: MSCWP
   ENO (TIB);
```

Figure 7-8. Bootstrap Microcode/Software Instructions. (2 of 2)

The primitive software bootstrap loaded from track 1 now begins execution. It loads in track Ø of the floppy. The execution of track Ø (which was just loaded) starts the loading of the operating system. Segments Ø,2,3, and 5 of the operating system are loaded into upper memory. These segments contain the I/O drivers for the operating system. At this point, the operating system starts execution of segment 5, which performs I/O initialization. Then segment 6 is loaded into upper memory, and the operating system command prompt appears. The operating system is now ready to accept user commands.

7.1.3 Registers and Operating System Tables

All registers are referenced by register number. The available registers and their numbers are as follows:

-3	Ready Queue Pointer [RQP]
-2	Segment Vector Pointer [SDP]
-1	Current Task Pointer [CTP]
2	Lower Stack Pointer Limit [SPLOW]
3	Upper Limit of Stack [SPUPR]
4	Top of Stack Pointer [SP]
5	Active Mark Stack Control Word Pointer [MP]
6	Base Addressing Mark Stack Control Word Pointer [BP]
7	Program Counter [IPC]
8	Pointer to currently executing code segment [SEGB]

Registers are initialized in two ways. The first method is by the boot sequence. (Refer to section 1.7.2, the Bootstrap Sequence for details.) The second method is by the PMACHINE statement, a III.Ø UCSD Pascal language extension that allows generation of Pascal operators. For example, the program segment below reads the value of the markstack pointer into a Pascal variable.

```
CONST MP=5
      LPR=157; STO=196;

VAR   IMP: INTEGER;

BEGIN
  PMACHINE(^IMP,(MP),LPR,STO);
END.
```

A complete description of all the III.Ø UCSD Pascal operators is found in Appendix C.1 of this manual. The '^' in the PMACHINE statement places the address of the following identifier on top of the stack. An identifier (or expression) enclosed in parentheses is evaluated, and the result is placed on top of the stack. An expression without an '^' or parenthesis must be a constant and is placed directly into the code.

| NOTE |

The positive register numbers refer to values in the active TIB (Task Information Block). The Pascal declaration for the TIB is as follows:

```
TIB = RECORD { TASK INFORMATION BLOCK }
          REGS: PACKED RECORD
                  WAITQ: TIBP; { QUEUE LINK FOR SEMAPHORES }
                  PRIOR: BYTE; { TASK'S CPU PRIORITY }
                  FLAGS: BYTE; { STATE FLAGS...NOT DEFINED YET }
                  SPLOW: ^INTEGER; { LOWER STACK POINTER LIMIT }
                  SPUPR: ^INTEGER; { UPPER LIMIT ON STACK }
                  SP: ^INTEGER; { ACTUAL TOP-OF-STACK POINTER }
                  MP: MSCWP; { ACTIVE PROCEDURE MSCW PTR }
                  BP: MSCWP; { BASE ADDRESSING ENVIRONMENT PTR }
                  IPC: INTEGER; { BYTE PTR IN CURRENT CODE SEG }
                  SEGB: ^CODESEG; { PTR TO SEG CURRENTLY RUNNING }
                  HANGP: SEMP; { WHICH TASK IS WAITING ON }
                  IORSLT:IORSLTWD; {IO RESULT NEW IN TIB}
                  SIBS: ^SIBVEC { ARRAY OF SIBS FOR 128..255 }
               END { REGS } ;
          MAINTASK: BOOLEAN;
          STARTMSCW: MSCWP
          NEXTTIB: TIBP;
        END { TIB } ;
```

For example, the MP, the markstack control word pointer, is register number 5, and it is word 5 in the TIB. When the microcoded P-code operators LPR and SPR refer to positive-valued registers, these values are taken from the TIB.

The Segment Vector Pointer register points at the segment vector, which is an operating system table which contains information concerning all active segments in the Pascal System. The Pascal declaration for the segment vector is:

```
SEGVEC = ARRAY[Ø..127] OF SIBP
```

SIBP, (Segment Information Block pointer) is a pointer to a record containing information about each active segment. The Pascal declarations for SIB and SIBP are as follows:

```
SIBP = ^SIB;

SIB = RECORD { SEGMENT INFO BLOCK }
        SEGBASE: ^CODESEG;  { MEMORY ADDRESS OF SEG }
        SEGLENG: INTEGER;   { # WORDS IN SEGMENT }
        SEGREFS: INTEGER;   { NUMBER OF ACTIVE CALLS }
        SEGADDR: INTEGER;   { ABSOLUTE DISK ADDRESS }
        SEGUNIT: UNITNUM;   { PHYSICAL DISK UNIT }
        PREVSP:  INTEGER;
      END { SIB } ;
```

7.1.4 Concurrency Primitives and Interrupts

UCSD Pascal provides several language constructs that are useful for operating system and I/O handler development. Those constructs pertinent to intertask communication, and I/O coordination are described in the subsequent subsections.

Concurrency

The START command is the system intrinsic that creates new tasks in the system. This intrinsic may only be called from a main task, such as the outer block of a user program. If START is called from a subtask, a run-time error is generated. As a part of the START calling sequence, the semaphore primitives SIGNAL and WAIT are executed. The purpose of this semaphore synchronization for START is to assure that parameters passed by a START call are received by the subtask, before later execution may alter them. A user should note that this type of task switch occurs as a part of task STARTing. Calls to READ and WRITE execute the WAIT semaphore operator so a task switch may occur during I/O. Thus, a user should realize that a tasks switch may occur at other times than he has explicitly programmed using SIGNAL and WAIT.

A correct concurrent program makes no assumptions about the order of operations during concurrent processing. The corollary is that a program must always be prepared for a task switch because interrupts may happen at any time. In order to protect indivisible operations, a semaphore lock must be used.

| NOTE |

> A program that does not call START need not be concerned about concurrency and task switching as the operation for a single task handles all intertask synchronization.

I/O locks and associated critical regions are implemented at the unit level. These semaphore locks are used to assure that each I/O operation is not interrupted until it completes. This precaution assures that each UNITREAD, UNITWRITE, UNITCLEAR call is an indivisible operation for a specific unit and that no other task in the system may perform a unit operation on the same unit until the first operation completes.

Interrupts

Each time a hardware interrupt occurs, a software semaphore is signaled. When a hardware interrupt occurs, all interrupts are disabled. Thus, on receipt of an interrupt, a typical I/O driver, after checking status and capturing the I/O data, must reenable interrupts.

Interrupts may be enabled programatically. In order to enable interrupts, the interrupt enable register (at address FC48 hex for WD0900 and SB1600 systems and at FC41 for ME1600 systems) must be written to. For example, the Pascal procedure below enables interrupts.

```
procedure enableints;
var enabletrix: record
                  case boolean of
                    true:  (addr: integer);
                    false: (loc: ^integer);
                  end;
begin
  enabletrix.addr := -952;  { FC48 hex; and -959 (FC41) for ME1600 }
  enabletrix.loc^ := -1;
end;
```

For WD0900 machines, interrupts may be disabled by a program at the I/O device level. That is, each peripheral device (such as the WD 1931 on the serial port or the AM8255 on the parallel port) has a specific bit or bits that disable interrupts for the device. For example, to disable interrupts on a serial port, bits 1 and 2 of control register 1, the request to send and the receiver enable bits, must be reset.

For SB1600 systems, writing a 0 to FC48 disables interrupts and writing a 1 enables interrupts. For ME1600 systems, the mask registers at FC2C, FD2C, and FC42 can mask out interrupts. Refer to Appendix G for WD0900, SB1600, and ME1600 I/O addresses.

When an interrupt driven I/O driver executes, a typical sequence is:

```
Set up device controller registers to perform I/O
Wait(device interrupt semaphore)
Capture data
Reenable interrupts
```

In this sequence, the microcode handles the conversion of a hardware interrupt signal to a software signal. When an interrupt is generated by the hardware, interrupts are disabled for the entire system. The I/O driver sequence above reenables interrupts as soon as possible after the interrupt signal is received. Because of the necessity of reenabling interrupts after an I/0 interrupt, a guarantee must exist that when a hardware-interrupt-attached semaphore is signaled an I/O process is at a high enough priority that it executes in order to reenable interrupts.

When an I/O operation is requested by a program, the operating system (by use of signals to wait on semaphores) communicates with I/O handler concurrent tasks that act as managers for the hardware I/O resources. The I/O tasks are run at a priority between 240 and 255, so when a hardware interrupt occurs, the tasks can reenable system interrupts. In order to keep the I/O tasks at highest priority, no other task in the system may run at a higher priority. In order to safeguard this process, the START command does not allow a task to run at a priority higher than 240. If a task must be started at a higher priority, passing the stack space parameter as a negative number to the START command overrides this restriction.

Tasks

Tasks provide the basis for concurrent processing. A task is declared by a PROCESS declaration, which is a UCSD Pascal extension. The PROCESS declaration is syntactially similar to a PROCEDURE declaration.

```
Syntax:    Process <identifier> <formal parameter part>
           Process <identifier>
```

A process is started by the procedure Start described by the following format.

```
Syntax:    Start (<process statement>[<procesid var>],
                  [stacking expression>,
                  <priority expresion>]]]
```

Three optional parameters exist for the Start procedure.

1. Processid - a predeclared variable type in UCSD Pascal. When present, assigns a value to the variable which is unique to the process which has been started. In the example, each call of the process DDuck in Figure 7-9 has a different processid value. The first call does not return a value of processid because no parameter is passed.

2. Stacksize expression - determines how much stack space is allocated for this process. If no value is given, the compiler allocates a default value of 200 words.

3. Priority expression - determines what processes are handled first by the CPU. The higher priority processes are executed before lower priority processes. If no priority is given, the new process inherits the priority of the caller.

Figure 7-9 shows an example of the Start procedure.

```
Program cartoon;
varprocidl,procid2,procid3;
i,j : integer;

process mmouse;
   begin
   end
process dduck(x,y:integer);
   begin
   ...
   end;
begin  (*start of program cartoon*)
   start (mmouse);
   i = 1;
   j = 2;
   start (dduck(i,j));
   start (dduck(3,4),procidl);
   start (dduck(5,5),procid2,300);
   start (dduck(j,2),procid3,i+j,10);
```

Figure 7-9. Example of the Start Procedure.

Start commands can only be called from a main task such as the outer block of a user program. If called from a subtask, a run-time error is generated.

Each task has an associated TIB that reflects the status of each task. The Start procedure links a TIB created by the process into the ready queue in priority order. When a task is to execute, it is moved from the ready queue to the current task queue. This queue has only a single task on it: the one currently executing. The ready queue is manipulated by the semaphore primitives Wait and Signal.

Semaphores

Semaphores are an important part of concurrent tasking. They function in three areas.

- Solving mutual exclusion problems.
- Synchronization of timing between two cooperating programs.
- Attaching semaphores to a hardware interrupt enabling interrupt handlers to be written.

The internal structure of a semaphore consists of two elements.

- Count field - set by Seminit value.
- Queue field - indicating if any tasks are waiting on the semaphore. The queue field actually points to a linked list of TIBs waiting on the semaphore in priority order.

Semaphores are declared as a variable data type and must be initialized before being used. Failure to initialize a semaphore causes unpredictable actions from the system. Semaphores are initialized by the procedure Seminit.

Example:

```
Seminit (sem,Ø)
```

The two parameters associated with Seminit are (1) semaphore name (sem) and (2) an integer value that represents the initial count of the semaphore (in the example, the value is Ø). When the compiler encounters a Seminit, it sets the count part of the semaphore to the integer value of Seminit and the queue field to nil.

Signal and Wait

Signal and Wait use semaphore variables as parameters.

Wait -- When executing Wait on a semaphore, two possible paths exist depending on the count of the semaphore.

1. When the count of the semaphore is greater than zero, the count is decremented and the task continues.
2. When the count equals zero, the current task is enqueued on the semaphore. The next task on the ready queue is moved to the current task queue and executed.

Signal -- Two paths also exist for Signal when executing on a semaphore; however, in this case, the queue field is the deciding factor.

1. If the queue field is nil, the count field is incremented and the task continues.
2. When the queue field is not nil (that is, tasks are waiting on the semaphore), the task at the front of the semaphore queue is moved to the ready queue. No incrementing of the count field takes place. The highest priority task between the current task and those tasks in the ready queue then executes.

Figure 7-1Ø gives an example of the use of semaphores.

```
program semaphorexmple;
  var pid1,pid2 : processid;
  Messagelock, Messageready, Receivedmessage : semaphore;
  message : string;

  process Sendmessage (mess:string);
   {locals are allowed}
  begin
    wait (Messagelock);
    message := mess;
    signal (Messageready);
    wait (Receivedmessage);
    signal (Messagelock);
  end;  (*Sendmessage*)

  process Printmessage;
  begin
    wait (Messageready);
    writeln (message);
    signal (Receivedmessage);
  end;  (*Printmessage*)

  begin
    seminit (Messagelock,1);
    seminit (Messageready, Ø);
    seminit (Receivedmessage, Ø);

    start (Printmessage,pid1,85,2ØØ);
    start (Sendmessage('themessage'),pid2,85,2ØØ);

  end.
```

Figure 7-1Ø. Semaphore Example.

EXPLANATION OF SEMAPHORE EXAMPLE

Initial State

Current Task Queue	Ready Queue	Messagelock	Messageready	Receivedmessage
Printmessage	Sendmessage	1	Ø	Ø

Printmessage begins executing. The wait for (messageready) executes causing the printmessage task to be queued and the Sendmessage process to begin executing.

Current Task Queue	Ready Queue	Messagelock	Messageready	Receivedmessage
Sendmessage		1	Ø	Ø
			Que	
			Printmessage	

Sendmessage begins executing with a Wait (messagelock). This action decrements the messagelock count to zero, places the message passed into message (local variable) and then signals (Messageready). Because a task is waiting on the Messageready queue, this task is removed from the semaphore queue and placed on the ready queue.

Sendmessage continues executing and executes the wait (Receivedmessage). With the value now equal to zero, this state causes Sendmessage to be placed on the semaphore queue and Printmessage placed in the current task queue beginning execution from the last stopping point.

Current Task Queue	Ready Queue	Messagelock	Messageready	Receivedmessage
Printmessage		Ø	Ø	Ø
			Que	Que
				Sendmessage

Printmessage now outputs the message and signals (Receivedmessage), causing Sendmessage to be placed in the ready queue. Printmessage completes and Sendmessage resumes execution. Sendmessage signals (Messagelock) causing the count of Messagelock to be incremented. Sendmessage completes, and all semaphores are back to their initial states.

Attach

The intrinsic called Attach associates the semaphore parameter with an interrupt signal. Therefore, when the hardware raises the interrupt, the associated semaphore is signaled.

Example:

```
procedure attach(sem :semaphore,vector : integer);
```

Sem contains a pointer to the semaphore involved. Vector contains an interrupt address attached to that semaphore. Each I/O port on the ME16ØØ Series SuperMicro computer is a unique address.

7.2 INTRINSICS

Most of the standard functions described in the Pascal Users Manual and Report (2nd edition) by Kathleen Jensen and Niklaus Wirth (Springer-Verlag, 1975) are provided in the III.Ø Operating System. Many of these functions are described in the following sections; the standard functions provided but not described in these sections are as follows:

> NEW, ABS, SQR, SIN, COS, ARCTAN, EXP, LN, SQRT, ODD, TRUNC, ROUND, ORD, CHR, SUCC, and PRED.

The standard functions not provided are as follows:

> DISPOSE, PACK, and UNPACK.

Users of the UCSD Pascal(TM) III.Ø intrinsics should be fluent in Pascal and experienced in the use of the III.Ø Operating System. All necessary range and validity checks during use of those intrinsics are the responsibility of the user. Some intrinsics do no range checking. Those intrinsics that are particularly dangerous are noted in the descriptions.

The required parameters are listed along with the function/procedure identifier. Optional parameters are in square brackets []. The default values are in braces {} on the line below. Within each subsection, functions and procedures are listed in alphabetical order.

The following terms are used in the explanations of the Intrinsics:

ARRAY	: an ARRAY OF CHARacters
BLOCK	: one disk block, (512 bytes)
BLOCKS	: an INTEGER number of blocks
BLOCKNUMBER	: an absolute disk block address
BOOLEAN	: any BOOLEAN value

CHARACTER	: any expression that evaluates to a character
DESTINATION	: a string or PACKED ARRAY OF CHARacters into which to write or a STRING that is context dependent
EXPRESSION	: part or all of an expression to be specified
FILEID	: a file identifier that must be VAR fileid: FILE OF <type>; or TEXT; or INTERACTIVE; or FILE;
INDEX	: an index into a STRING or PACKED ARRAY OF CHARacters that is context dependent or as specified
NUMBER	: a literal or identifier whose type is either INTEGER or REAL
RELBLOCK	: a relative disk block address (relative to the start of the file in context); the first block being block zero
SIMPLE VARIABLE	: any declared PASCAL variable that is of one of the following TYPEs: BOOLEAN CHAR REAL STRING or PACKED ARRAY [..] OF CHAR
SIZE	: an INTEGER number of bytes or characters; any integer value
SOURCE	: a STRING or PACKED ARRAY OF CHARacters to be used as a read-only array, context dependent or as specified**
SCREEN	: an array 80 X 24 bytes long; or as needed
STRING	: any STRING, call-by-value unless otherwise specified; that is, may be a quoted string, string variable, or function that evaluates to a STRING
TITLE	: a STRING consisting of a file name
UNITNUMBER	: physical device number (See Appendix B.4.)
VOLID	: a volume identifier; STRING [7]

** In string intrinsics, SOURCE must be a string. In intrinsics that deal with packed arrays of characters, it may be either. However, in using STRINGs in intrinsics that expect character arrays, the zero element of the string is the length byte, which may cause some unexpected problems if not previously considered.

7.2.1 Character Array Manipulation Intrinsics

The Character Array Manipulation Intrinsics are byte-oriented. No range checking of any kind is performed on the parameters passed to them; therefore, caution must be used in dealing with these intrinsics. The system does not protect itself from these operations. The intrinsic SIZEOF (Section 7.2.4) is intended for use with these intrinsics when the number of bytes is a parameter.

PROCEDURE FILLCHAR (DESTINATION, LENGTH, CHARACTER);

This procedure takes a (subscripted) packed array of characters and fills it with the number (LENGTH) of CHARACTERs specified. This same action can be done using a MOVELEFT procedure (described below); but the FILLCHAR procedure is twice as fast because no memory reference is needed for the source.

PROCEDURE MOVELEFT (SOURCE, DESTINATION, LENGTH);
PROCEDURE MOVERIGHT (SOURCE, DESTINATION, LENGTH);

These procedures do mass moves of bytes for the LENGTH specified. (The LENGTH is in bytes.) MOVELEFT starts from the left end of the SOURCE and moves bytes to the left end of the DESTINATION, traveling right. MOVERIGHT starts from the right end, traveling left. Both may be needed when working on a single array in which the order of the characters moved is critical.

See Figure 7-11 for an example use of the MOVELEFT and MOVERIGHT procedures.

```
PROGRAM MOVETEST;
VAR BUF1  :  PACKED ARRAY [Ø..19] OF CHAR;
    BUF2  :  PACKED ARRAY [Ø..2Ø] OF CHAR;

BEGIN (*MOVETEST*)
  BUF1  := 'MOVE CHARACTERS LEFT';
  BUF2  := 'THESE CHARACTERS.....';
  MOVELEFT(BUF1,BUF2,5); {move 5 bytes from BUF1 to BUF2 going L to R}
  WRITELN (BUF2);
END (*MOVETEST*).
```

Figure 7-11. Example of the MOVELEFT, MOVERIGHT Character Array Manipulation Intrinsic.

FUNCTION SCAN (LENGTH, PARTIAL EXPRESSION, ARRAY) : INTEGER;

This function returns the number of characters from the starting position of the scan to the position where it terminated. Termination comes when matching the specified LENGTH or satisfying the EXPRESSION. The ARRAY should be packed and may be subscripted to denote the starting point. If the EXPRESSION was satisfied on the character at which ARRAY is pointed, the value returned is zero. If the LENGTH passed was negative, the number returned is negative, and the function will have scanned backward. The PARTIAL EXPRESSION must be in the following format:

"<>" or "=" followed by character expression.

See Figure 7-12 for an example use of SCAN.

```
PROGRAM SCANTEST;
VAR EX : PACKED ARRAY[Ø...37] OF CHAR;
    I  :  INTEGER;

BEGIN (*SCANTEST*)
  EX := ' EXAMPLE OF CHARACTER ARRAY INTRINSICS';
  I  := SCAN(-25,=':',EX[25]);   {starting at 25th char in EX, scan}
  WRITELN (I);                   {to the left until a : is found; or}
  I  := SCAN(38,<>' ',EX{Ø});    {until the end is reached.}
  WRITELN (I);
END (*SCANTEST*).
```

Figure 7-12. Example of the Scan Character Array Manipulation Intrinsic.

7.2.2 I/O Intrinsics

```
FUNCTION BLOCKREAD(FILEID,ARRAY,BLOCKS,[RELBLOCK]) : INTEGER;
FUNCTION BLOCKWRITE(FILEID,ARRAY,BLOCKS,[RELBLOCK]) : INTEGER;
                                        {SEQUENTIAL}
```

These functions return an INTEGER value of the number of blocks of data transferred. The FILE must be an untyped file. The length of ARRAY should be an integer multiple of bytes-per-disk-block. BLOCKS is the number of blocks to be transferred. RELBLOCK is the blocknumber relative to the start of the file, block zero being the first block. If no RELBLOCK is specified, the I/O is completed sequentially starting at block zero. A random access I/O moves the file pointers. EOF(FILEID) becomes true when the last block in the file is read.

```
PROCEDURE CLOSE (FILEID, [OPTION]);
```

OPTIONS include ", LOCK", ", NORMAL", ", PURGE" and ", CRUNCH". (The commas must appear as shown; that is, the option must be preceded by a comma.)

A normal CLOSE is done when the OPTION is null. Normal means the following: if open with reset, then CLOSE leaves the file in the directory; if open with rewrite, then CLOSE purges the file. CLOSE simply sets the file state to closed. If the file is a disk file and was opened using REWRITE, it is deleted from the directory.

The LOCK option causes the disk file associated with the FILEID to be made permanent in the directory if the file is on a directory-structured device and the file was opened with a REWRITE; otherwise, a normal CLOSE is done.

The PURGE option deletes the title associated with the FILEID from the directory. The unit goes off-line if the device is not block-structured.

The CRUNCH option locks the file with the current location of the file pointer being the last record of the file.

| CAUTION |

If a SEEK has been done on the file, the file pointer may not point to the end of the file. The records after the file pointer are discarded.

Regardless of OPTION, all CLOSEs mark the file closed and make the implicit variable FILEID^ undefined. CLOSEing an already CLOSEd file causes no action.

FUNCTION EOF (FILEID) : BOOLEAN;
FUNCTION EOLN (FILEID) : BOOLEAN;

EOF (end-of-file) and EOLN (end-of-line) return False after the file specified is reset. They both return True on a closed file. If FILEID is not present, the fileid INPUT is assumed (for example, IF EOF THEN. . .). When EOF (FILEID) is True, FILEID^ is undefined.

When GET (FILEID) sets FILEID^ to the EOLN or EOF character, EOLN (FILEID) returns True, and FILEID^ (in a FILE OF CHAR) is set to blank.

While doing PUTs or WRITEs at the end of a file, if the file cannot be expanded to accommodate the PUT or WRITE, EOF (FILEID) returns True.

PROCEDURE GET (FILEID);
PROCEDURE PUT (FILEID);

GET (FILEID) leaves the contents of the current logical record pointed at by the file pointers in the implicitly declared window variable FILEID^ and increments the file pointer.

PUT (FILEID) puts the contents of FILEID^ into the file at the location of the current file pointers and then updates those pointers.

Both procedures are used on typed files; that is, files for which a type is specified in the variable declaration ("FILEID: FILE OF type"). Untyped files are simply declared as "FILEID: FILE;". "F: FILE OF CHAR" is equivalent to "F: TEXT". In a typed file, each logical record is a memory image fitting the description of a variable of the associated <type>.

FUNCTION IORESULT : INTEGER;

After any I/O operation, IORESULT contains an INTEGER value that represents the error result (a Ø means no error). Refer to Appendix B3 for a list of error results.

PROCEDURE PAGE (FILEID);

PAGE (FILEID) sends a top-of-form (ASCII FF) to the file.

PROCEDURE READ{LN} (FILEID, SOURCE);
PROCEDURE WRITE{LN} (FILEID, SOURCE);

These procedures may be used only on TEXT (FILE OF CHAR) or INTERACTIVE files for I/O. Three predeclared INTERACTIVE files are available for use: INPUT, OUTPUT, and KEYBOARD. INPUT results in echoing of characters typed to the console. OUTPUT allows the user to halt or flush the output by use of START/STOP and FLUSH characters. (See the discussion of SETUP, 6.1.) KEYBOARD does no echo; it allows the programmer complete control of the response to user typing.

If "FILEID," is omitted, INPUT or OUTPUT (as appropriate) is assumed. A READ (STRING) reads up to, but not including, the end-of-line character (carriage return) and leaves EOLN (FILEID) True. This action means that any subsequent READs of string variables return the null string until a READLN or READ (character) is executed.

PROCEDURE RESET (FILEID, [TITLE]);
PROCEDURE REWRITE (FILEID, TITLE);

These procedures open files for reading and writing and mark the file as open. The FILEID may be any Pascal-structured file. TITLE is a string containing any legal file title. REWRITE creates a new file on disk for output files; RESET marks an already existing file open for I/O. For both, RESET and REWRITE, if the device specified is a non-directory-structured device (for example, REMOTE:), the file is opened for input, output, or both.

If the file is already open when the RESET (with TITLE) or REWRITE is attempted, an error is returned in IORESULT. The state of the file remains unchanged.

RESET (FILEID) without an optional parameter applied to an already open file rewinds the file by setting the file pointers back to the beginning (record Ø) of the file.

On INTERACTIVE files, RESET does not GET the file. On all other types of files RESET does an initial GET on the file, setting the window variable to the first record in the file.

REWRITE allows use of file size specification in the title, consisting of "[<number of blocks>]" at the end of the title string. The size specification affects the location of the disk space for the file; it does not determine the size of the file.

Examples:

```
RESET(FILEID, STRINGID); {opens STRINGID for input}
REWRITE(FILEID,'VOLUME:FILE.TEXT[4]'); {opens VOLUME.FILE.TEXT}
                                        {for output creating a}
                                        {file 46 blocks long.}
```

PROCEDURE SEEK (FILEID, INTEGER);

SEEK changes the file pointers so that the next GET or PUT uses the INTEGERth record of FILEID. Records in files are numbered starting with Ø. A GET or PUT must be executed after a SEEK call before the window and associated buffers are valid.

FUNCTION UNITBUSY (UNITNUMBER) : BOOLEAN;

This function returns a Boolean value. If the value is True, the device specified is actively performing an I/O transfer. For example:

```
IF NOT UNITBUSY(1) THEN
  WRITELN('Please type a character');
```

Execution of the example results in the output of the line 'Please type a character' until a character has been typed. For the units CONSOLE: and REMOTE:, UNITBUSY returns True if characters exist in the typeahead queue.

PROCEDURE UNITCLEAR (UNITNUMBER);

This procedure cancels all I/O requests to the specified unit and resets the hardware to its power-up state.

PROCEDURE UNITREAD (UNITNUMBER, ARRAY, LENGTH, [BLOCKNUMBER], [FLAGS]);
PROCEDURE UNITWRITE (UNITNUMBER, ARRAY, LENGTH, [BLOCKNUMBER], [FLAGS]);

These procedures are dangerous because no range checking is done.

These low-level procedures perform I/O to various devices.

The UNITNUMBER is the integer name of the device. ARRAY is any declared packed array. It may be subscripted to indicate a starting position from or to which the transfer is to be completed. LENGTH is an integer giving the number of bytes to transfer.

BLOCKNUMBER is required only when using a block-structured device, and is the absolute block number from or to which the transfer is to complete. The FLAGS value is optional. Bit Ø of the FLAGS value set implies asynchronous I/O. (See the following discussion of asynchronous I/O.) Bit Ø reset implies synchronous I/O. (This bit should always be reset.)

Bit 1 of FLAGS reset implies logical sector mode, which is the normal mode on the system. If bit 1 is set, physical sector mode is enabled. This mode has the effect that BLOCKNUMBER is interpreted as the physical sector number. Conceptually in this mode, the disk looks like an array of tracks where each track is an array of sectors. Physical sectors are numbered from Ø to 25 starting on track Ø, continue ascending on side 1, track Ø, if it exists and then on to track 1, side Ø, and so forth. For single-sided, single-denity diskettes, track 1 has sectors 26-51 (sector size is 128 bytes). For single-sided, double-density diskettes, track 1 has sectors 26-51 (sector size is 256 bytes). For double-sided, single-density diskettes, track 1 has sectors 51-77 (sector size is 128 bytes). For double-sided, double-density diskettes, track 1 has sectors 51-77 (sector size is 256 bytes). This mode is especially useful for accessing track Ø of a diskette, where the bootstrap resides. For example, the following code sequence reads all of track Ø into an array:

```
VAR TRACKBUF: ARRAY[Ø..3327] OF Ø..255;

UNITREAD(4,TRACKBUF,3328,Ø{sector Ø},2{physical mode});
```

|NOTE|

> Track Ø, side Ø is always single density, even if the diskette is a double-density diskette.

For Winchester drives, a sector contains 512 bytes (it is the same size as a block). Track Ø, head Ø contains sectors Ø..15; track Ø, head 1 contains 16..31, and so forth.

Bit 2 of FLAGS set implies no special character handling of DLEs, the blank compression code, or the EOF character.

Bit 3 of FLAGS set implies no line feeds are appended to carriage returns.

All of these values are normally reset. If BLOCKNUMBER is omitted, but FLAGS is included, a comma is used to hold the placement of parameters.

Asynchronous I/O

With an H-level interrupt driven operating system, the technique of asynchronous I/O as implemented in some UCSD II.Ø systems can be simulated using the tasking constructs of the III.Ø Operating System. The program in Figure 1-13 is an example of asynchronous I/O simulation.

```
program asynch;
var ch: char;
    gotchar: boolean;

process reader;
begin
  read(ch);
  gotchar := true;
end;

begin
  ch := 'i'; gotchar := false;
  start(reader,,1ØØ,15Ø); { Priority 15Ø higher than main task }
  while not gotchar do
    writeln('Please type a character');
  writeln(ch,'   was typed');
end.
```

Figure 7-13. Example of Asynchronous I/O Simulation.

PROCEDURE UNITWAIT (UNITNUMBER);

The program or task that executes this statement waits on the unit until the specified unit is not actively performing an I/O transfer. This wait is implemented using locking semaphores to guard each unit I/O operation.

7.2.3 String Intrinsics

To maintain the integrity of the LENGTH of a string, only string functions or full-string assignments should be used to alter strings. Moves and single-character assignments do not affect the length of a string; therefore, the programmer must do range checking. The individual elements of STRING are of CHAR type and may be indexed 1. . LENGTH(STRING). Accessing the string outside this range has unpredictable results if range-checking is off. If range-checking is on, a run-time error results.

Examples of String Intrinsics are given in Figure 7-14.

FUNCTION CONCAT (SOURCEs) : STRING

> This function returns a string that is the concatenation of all the strings passed to it. Any number of source strings, separated by commas, may exist.

FUNCTION COPY (SOURCE, INDEX, SIZE) : STRING

> This function returns a string containing SIZE characters copied from SOURCE starting at the INDEXed position.

FUNCTION LENGTH (STRING) : INTEGER

> This function returns the integer value of the length of STRING.

FUNCTION POS (STRING, SOURCE) : INTEGER;

> This function returns the integer position of the first occurrence of the pattern (STRING) in SOURCE. If the pattern was not found, zero is returned.

PROCEDURE DELETE (DESTINATION, INDEX, SIZE);

> This procedure deletes SIZE characters from DESTINATION starting at the INDEXed position.

PROCEDURE INSERT (SOURCE, DESTINATION, INDEX)

> This procedure inserts SOURCE into DESTINATION starting with the INDEXed position in DESTINATION.

PROCEDURE STR(LONG,DESTINATION);

> This procedure converts a long integer LONG into a string. The resulting string is placed in DESTINATION. The integer LONG may also be a normal INTEGER.

```
PROGRAM STRINTST;

USES longint;
VAR name,text,pattern,first,second,third : STRING;
    start,get,toomany,more : STRING;
    long : INTEGER[8];
    I : INTEGER;

BEGIN (*STRINTST*)

 I := LENGTH('ABC');
 WRITELN (I);
 name := 'JOHN SMITH';

 I := LENGTH(name);
 WRITELN(I);
 text := 'THIS IS AN EXAMPLE OF STRING INTRINSIC';
 pattern := 'EXA';

 I := POS(pattern,text);
 WRITELN(I);

 first := 'ABCDE';
 second :='FGHIJ';
 third := CONCAT(first,second);
 WRITELN (third);

 start := 'HERE IS A STRING OF CHARACTERS';
 get := COPY(start,POS('C',start),10);
 WRITELN(get);

 toomany :='THIS STRING HAS TOO MANY CHARACTERS';
 DELETE(toomany,17,9);
 WRITELN(toomany);

 more :=' TOO MANY';
 INSERT(more,toomany,16);
 WRITELN(toomany);

 long := 1000000;
 STR(long,more);

 WRITELN('$',more);
END(*STRINTST*).
```

Figure 7-14. Examples of String Intrinsics

7.2.4 Miscellaneous Intrinsic Routines

PROCEDURE GOTOXY (XCOORD, YCOORD);

This procedure sends the cursor to the specified coordinates. The upper left corner of the screen is assumed to be Ø,Ø. This procedure defaults to a Volker-Craig VC44Ø4 terminal. For systems using another terminal, a new GOTOXY must be bound in (see Section 6.1Ø).

PROCEDURE HALT;

This procedure generates an opcode that causes a run-time error to occur.

FUNCTION LOG (NUMBER) : REAL;

This function returns the log base ten of NUMBER.

PROCEDURE MARK (VAR HEAPPTR: ^INTEGER);
PROCEDURE RELEASE (VAR HEAPPTR: ^INTEGER);

These procedures allocate and return heap memory space to the system. HEAPPTR is of type ^INTEGER. MARK sets HEAPPTR to the current top-of-heap. RELEASE sets the top-of-heap pointer to HEAPPTR.

FUNCTION MEMAVAIL: INTEGER;

This function returns the available space as the number of words between the top of the stack and the top of the heap. On a 128K-byte system , MEMAVAIL may be greater than 32K-words. Any integer greater than 32K (32767) words is represented as a negative number. In order to avoid failure of MEMAVAIL tests if the space available is greater than 32K words, MEMAVAIL returns 32767 if true available memory is greater than that number. REMAVAIL returns the exact size.

FUNCTION PWROFTEN (EXPONENT: INTEGER) : REAL;

This function returns the value of ten to the EXPONENT power. EXPONENT must be an integer in the range of Ø through 37.

FUNCTION RMEMAVAIL: REAL;

This function returns a real value that exactly represents the true memory space available, regardless of whether the system is configured with 64K or 128K bytes.

FUNCTION SIZEOF (VARIABLE OR TYPE IDENTIFIER): INTEGER;

This function returns the number of bytes that a parameter occupies in the stack. SIZEOF is particularly useful with the FILLCHAR and MOVExxxx intrinsics.

PROCEDURE TIME (VAR HIWORD, LOWORD: INTEGER);

This procedure returns the current value of the system clock in 60ths of a second. The HIWORD contains the most significant portion, and LOWORD contains the least significant portion. Both HIWORD and LOWORD must be VARiables of type INTEGER. This procedure is meaningful only on ME1600 systems that have a real-time clock.

7.2.5 Concurrency and Interrupt Intrinsics

See Section 7.1.4 Concurrency Primatives and Interrupts for further details.

PROCEDURE ATTACH(SEMAPHORE,INTEGER);

This procedure attaches the semaphore to the interrupt address specified by the integer, allowing a hardware interrupt to signal a semaphore.

PROCEDURE SEMINIT(SEMAPHORE,INTEGER);

This procedure initializes the semaphore. The integer value specifies the number of times the semaphore has been signaled. The following example initializes the semaphore SEM to "not signaled".

```
SEMINIT(SEM,0);
```

PROCEDURE SIGNAL(SEMAPHORE);

If any tasks are waiting on the semaphore, the first task on the semaphore queue is moved to the ready queue (in priority order); the task with the highest priority among the current task and those in ready queue then executes.

If no tasks are waiting on the semaphore, its number of outstanding signals is incremented, and the current task continues to execute.

PROCEDURE START(PROCESS(PARAMS),PROCESSID,INTEGER,INTEGER);

This procedure causes the process to be initiated asynchronously. The processid is assigned to point to the TIB that is initialized. The two integer parameters, STACKSPACE and PRIORITY, respectively, specify the amount of stack space the task is allocated and the priority at which it runs. PRIORITY is of type Ø..255.

| NOTE |

Priorities 24Ø-255 are reserved for operating system I/O drivers. The highest priority available to user programs is 239.

(See 7.1.3 Registers and Operating System Tables for a description of the TIB.) The highest priority task not waiting on a semaphore executes at the conclusion of the START.

PROCEDURE WAIT(SEMAPHORE);

If the semaphore has already been signaled, its number of outstanding signals is decremented, and the current task continues to execute.

If the semaphore has not been signaled, the current task is moved to the semaphore queue in priority order, and the highest priority task in the ready queue executes. In this case, if the ready queue is empty, the processor waits for an I/O interrupt to occur.

The example in Figure 7-15 illustrates the use of the WAIT procedure described above.

```
program ProcessExample;
var pid1,
    pid2: processid;
    MessageLock,
    MessageReady,
    ReceivedMessage : semaphore;
    Message : string;

process SendMessage(mess : string);
  {locals are allowed}
begin
  wait(MessageLock);
  message:= mess;
  signal(MessageReady);
  wait(ReceivedMessage);
  signal(MessageLock);
end;  {SendMessage}

process PrintMessage;
begin
  wait(MessageReady);
  writeln(message);
  signal(ReceivedMessage);
end;  {PrintMessage}

begin
  seminit(MessageLock,1);
  seminit(MessageReady,Ø);
  seminit(ReceivedMessage,Ø);

  start(PrintMessage,pid1,85,2ØØ);
  start(SendMessage('The message'),pid2,85,2ØØ);

end.
```

Figure 7-15. Example of WAIT Procedure.

7.3 SEGMENTS

Segmenting a program so that procedures must be in memory only when required has many advantages:

- Large pieces of one-time code (for example, initialization procedures) can be put into a segment.
- A program can be configured to suit storage requirements.

A maximum of 128 user segments are available. These segments are numbered 128..255. Also, nine system segments (1, 8..15) are available for user programs.

The disk that holds the code file for the program must be on line and in the same drive as when the program was started whenever a SEGMENT is called. A message requesting the correct disk is generated.

SEGMENT procedures must be the first procedure declarations containing code-generating statements. Declarations of SEGMENT procedures and functions in UCSD Pascal are identical to those in standard Pascal, except that they are preceded by the reserved word "SEGMENT".

As an example, when the user wishes to put initialization procedures into a segment because they are one-time-only procedures, the declaration might be:

```
SEGMENT PROCEDURE INITIALIZE;
BEGIN
  (* Pascal code *)
END;
```

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

7.4 LINKAGES

Frequently used routines and data structures can be separately compiled and can be stored in libraries until needed (see Section 6.5, Librarian). These externally compiled structures can be integrated into files that need to use them. A file that references such a structure need not compile it directly into its code file; the Linker (see Section 6.7) copies the existing code of the structure into the host code file.

Separate compilation, followed by linkage when needed, is supported by the III.Ø Operating System between portions of programs written in Pascal, as described below.

7.4.1 Pascal-to-Pascal Linkages (Units)

A UNIT is a Pascal routine composed of interdependent procedures, functions, and associated data structures. Whenever the routine is needed within a Pascal program, the program USES the UNIT.

A UNIT has two parts. The INTERFACE part declares constants, types, variables, procedures, and functions that are public and may be used by the host program. In other words, the INTERFACE part defines how the host program communicates with the UNIT. The IMPLEMENTATION part declares the same types of items; however, these items are private to the UNIT and are not available to the host program. Also, the IMPLEMENTATION part defines how the UNIT accomplishes its task.

When the Compiler encounters a USES statement, it references the INTERFACE part of the UNIT as though that part belonged to the host program itself. Because the constants, types, variables, functions, and procedures declared in the INTERFACE part are global, name conflicts may arise with identifiers in the host program. The programmer may not declare global identifiers with the same name as used in the INTERFACE part of a UNIT. Procedures and functions may not USE UNITs locally.

Figure 7-16 shows an example of USES.

```
program writedate;
uses screencontrol;
var mo : months;
    days : days;
    yr : years;
begin
  home;
  cleareos;
  date(mo,day,yr);
  writeln('The date is: ', mo:2, '-', day:2, '-', yr:2);
end.
```

Figure 7-16. Example of USES.

The syntax for a UNIT definition is shown in Figure 7-17. See Section 7.5.1 for the interface of SCREENCONTROL. The declarations of routine headings in the INTERFACE part are similar to forward declarations; therefore, when the corresponding routines are defined in the IMPLEMENTATION part, formal parameter specifications cannot be repeated.

| NOTE |

Variables of type FILE must be declared in the INTERFACE part of a UNIT. A FILE declared in the IMPLEMENTATION part causes a syntax error at compile time.

```
<Compilation unit>        ::= <Program heading>;{<Unit definition>;}
                              <Uses part> <Block> |
                              <Unit definition>;{<Unit definition>}.

<Unit definition>         ::= <Unit heading>;
                              <Interface part>
                              <Implementation part>
                              End

<Unit heading>            ::= Unit <Unit identifier>

<Unit identifier>         ::= <Identifier>

<Interface part>          ::= Interface
                              <Uses part>
                              <Constant definition part>
                              <Type definition part>
                              <Variable declaration part>
                              <Procedure and Function heading part>

<Procedure and function heading part>
                          ::= {<Procedure or function heading>}

<Procedure and function heading>
                          ::=<procedure heading>|<function heading>

<Implementation part>     ::= Implementation
                              <Label declaration part>
                              <Constant definition part>
                              <Type definition part>
                              <Variable declaration part>
                              <Procedure and Function declaration part>

<Uses part>               ::= Uses <Unit identifier>
                              {, <Unit identifier>}; | <Empty>
```

Figure 7-17. Syntax for a UNIT Definition.

A user may define a UNIT in-line, after the heading of the host program. In this case, the user compiles both the UNIT and the host program together. Subsequent changes in either the UNIT or host program require a recompilation of both.

The Linker copies the code for the UNIT into the host program.

A UNIT or group of UNITs can be compiled separately and stored in a library. After compiling a host program that uses a UNIT stored in a library, the user must link that UNIT into the host program by executing the Linker. If a user calls R(un and an unlinked code file is requested, the Linker is called automatically. If X(ecute is called in such a case, the system issues a reminder to link the code.

If the host program has changes, the user must recompile and link in the UNIT. If the IMPLEMENTATION part is changed, the UNIT must be recompiled, and then all compilation units that use the UNIT must be relinked. Changes in the INTERFACE part require a recompilation of not only the UNIT, but of all compilation units that use it. Then, all compilation units must be relinked. These restrictions apply only if the new version of the UNIT is to be used in all files.

The Compiler generates Linker information in the contiguous blocks that follow a program that uses UNITs. This information includes locations of references to externally defined identifiers.

7.5 SYSTEM LIBRARY

SYSTEM.LIBRARY contains five units: SCREENCONTROL, LONGINT, MENU, KBDSTUFF, and DELAYUNIT. The SCREENCONTROL unit contains several procedures that cause screen control action or that return information about the user's terminal. The LONGINT unit is used by code files that use long integers. The MENU unit allows menus to be developed to aid the end user in using the system. The KDBSTUFF unit allows a command file to be created which can be used to cause certain actions (for example, executing a program by a call from another program). The DELAYUNIT allows scheduled delays to suspend the task requesting a delay.

7.5.1 SCREENCONTROL Unit

The SCREENCONTROL unit accesses fields in the record SYSCOM, which is set up at boot time from the file SYSTEM.MISCINFO, which is created by the program SETUP. This unit contains several procedures that cause screen control action, return information about the terminal, or return the date. The user may access the following procedures in this unit.

PROCEDURE HOME;	{Homes the cursor using the characters specified in SETUP}
PROCEDURE CLEAREOS;	{Clears the screen starting at the current cursor position using the characters specified in SETUP}
PROCEDURE CLEAREOL;	{Clears the line starting at the current cursor position using the characters specified in SETUP}
FUNCTION SCREENWIDTH:INTEGER;	{Returns the width of the screen as specified in SETUP}
FUNCTION SCREENHEIGHT:INTEGER;	{Returns the height of the screen as specified in SETUP}

```
PROCEDURE DATE(VAR M:MONTHS; VAR D:DAYS; VAR Y:YEARS);
                       {Returns the current date as stored by the
                        operating system.  MONTHS, DAYS, and YEARS are
                        types declared in the INTERFACE and are, there-
                        fore, available to the user.  The declarations
                        are:    TYPE MONTHS = Ø...12;
                                     DAYS   = Ø...31;
                                     YEARS  = Ø...99;
```

7.5.2 LONGINT Unit

The optional use of a length attribute on the Pascal predeclared type INTEGER is available. An INTEGER with a length attribute is referred to as a LONG INTEGER. It is suitable for business, scientific or other applications where a need for extended number length with complete accuracy exists. The four basic standard arithmetic operations (addition, subtraction, multiplication and division) are supported, as well as routines facilitating conversion to strings and standard INTEGERs. Strong type checking is enforced to reduce potential errors. I/O, in-line declaration of constants, and inclusion in structured types are fully supported and are analogous to the usage of standard INTEGERs.

LONG INTEGERS are declared by using the standard identifier "INTEGER" followed by a length attribute enclosed in square brackets. The length is given as an unsigned number, not larger than 36, that denotes the minimum number of decimal digits to be represented. In the example below, the variable Z is capable of storing up to a 12-decimal digit signed number:

```
VAR Z:  INTEGER[12];
```

Generally, LONG INTEGERs may be used anywhere a REAL would be syntactically correct. However, care must be taken to ensure that sufficient words have been allocated by the declared length attribute for storage of the result of assignment or arithmetic expression statements. INTEGER expressions are implicitly converted as required on assignment to, or arithmetic operations with, a LONG INTEGER, but the reverse is not true. The LONG INTEGER cannot be used in a subrange, and conversion to type REAL is not supported.

| NOTE |

Long integers reside in the UNIT LONGINT and a USES LONGINT statement must be included in a program that performs long integer operations.

Examples of uses of the LONG INTEGER are shown in Figure 7-18.

```
PROGRAM LINTEGER;
 USES LONGINT;
 VAR L : INTEGER[20];
     I : INTEGER;
 BEGIN (*LINTEGER*)
  L := 9876543210;
  L := -L;
  L := L+L;
  L := 256;
  I := TRUNC(L);
 END (*LINTEGER*).
```

Figure 7-18. Example Uses of Long Integers.

Arithmetic operations that may be used in conjunction with the LONG INTEGER are as follows:

+, -, *, DIV, unary plus/minus

On assignment, the length of the LONG INTEGER is adjusted during execution to the declared length attribute of the variable. Therefore, an interrupt (overflow) can result when the intermediate result exceeds the number of words required to store at least 37 decimal digits, or when the final result is assigned to a variable with an insufficient length attribute. All of the standard relational operators can be used with mixed INTEGER and LONG INTEGER.

The function TRUNC accepts a LONG INTEGER as well as a REAL as an argument. Interrupt (overflow) occurs if the result is greater than MAXINT.

The procedure STR(L,S) converts the INTEGER or LONG INTEGER "L" into a string, complete with minus sign if needed, and places it in the STRING "S".

An attempt to declare a LONG INTEGER in a parameter list other than for the routines TRUNC and STR results in a compile-time error. The error may be circumvented by creating a type that is called LONG INTEGER, as follows:

```
TYPE LONG = INTEGER [15];
PROCEDURE OVERSIZE(ACCOUNT: LONG);
```

The LONG INTEGER is stored in a multiword, packed, binary-coded decimal (BCD) representation. System routines complete the I/O conversions as required. Maximum storage efficiency is achieved by dynamic expansion and contraction of word allocation as required. During LONG INTEGER operations, the length is placed on the stack above the number itself.

| NOTE |

> The declared length attribute is enforced (given range checking) only on assignment to a long INTEGER variable; an intermediate expression result can be up to 36 digits.

7.5.3 MENU Capability

The MENU unit allows creation of menus so that end users need not understand the details and intricacies of the III.Ø Operating System. This unit is valuable for applications developers in that it allows them to tailor the operating system to match the end user of the application.

Three procedures comprise the MENU unit.

MENUENABLE -->

This procedure enables or "turns on" the menu capability. That is, the program SYSTEM.MENU (this program is developed by the applications programmer) is executed. When SYSTEM.MENU is executed, the III.Ø Operating System command prompt line does not appear. The SYSTEM.MENU program controls the interface between the system and the end user. The end user, therefore, does not interface directly with the III.Ø Operating System.

MENUDISABLE -->

This procedure disables or "turns off" menu capability. The normal III.Ø Operating System command line appears.

CHAIN -->

This procedure allows one Pascal program to programmatically call another Pascal program. For example, CHAIN('Y.CODE') executed at the end of a program behaves as if it were interactively input from the user to X(ecute Y.

This form of chaining works with or without menus. The interaction between chaining and menus happens when menus are enabled because SYSTEM.MENU is executed until a chain command overrides the menu call for one execution of the chained program.

| NOTE |

A call to chain terminates the calling program because it contains an EXIT call.

The field to determine if menus are enabled or disabled is stored in SYSTEM.MISCINFO; the menus may be enabled/disabled by the SETUP program. (See Section 6.1.)

The example programs in Figure 7-19 illustrate SYSTEM.STARTUP and SYSTEM.MENU acting in conjunction to create a user menu interface.

```
program startup;
uses menu;
begin
  menuenable; { SYSTEM.STARTUP not needed if SETUP enabled menus in SYSCOM }
end.


program menu;
uses menu;
var ch : char;
begin
  repeat
    write('A(system, B(system, T(erminate menus'); read(ch); writeln;
  until ch in ['A','a','B','b','T','t'];
  case ch of
    'A','a': chain('a.code');
    'B','b': chain('b.code');
    'T','t': menudisable;
  end;
end.
```

Figure 7-19. Programs to Create User Menu Interface.

7.5.4 KBDSTUFF Unit

This unit allows the capability to create a command file. Up to 80 characters may be placed in the typeahead queue by this unit; also, a keyboard command file may be used to execute a program by a call from another program (chaining).

The UNIT KBDSTUFF provides a procedure whose Pascal declaration is:

```
PROCEDURE KBDBATCH(FUNIT: INTEGER; KSTRING: STRING);
```

A program may USE this UNIT by assigning interactive commands to KSTRING. Because a UCSD Pascal string cannot have embedded carriage returns, the ASCII character '~' is used to represent a carriage return in string input for keyboard commands. If any other nonprintable character is to be inserted into a string, a character assignment statement must be used. For example, the following program calls the editor and inserts the string 'ABC' at the start of the work file.

```
program commandeditor;
uses kbdstuff;
var s :string;
begin
  s := 'eiABCC';  { Last 'C' is a space holder }
  s[6] := chr(3); { ETX to terminate the insert }
  kbdbatch(1,s);
end.
```

The following program calls KBDBATCH, passing a command string that calls the Filer, requests a display of the volumes on line, lists the directories on the boot unit and on unit #5, and then asks what the work file is.

```
program commandfiler;
uses kbdstuff;
var s :string;
begin
  kbdbatch(1,'fvl*~ l#5~ w');
end.
```

Chaining is performed by the following sample program which contains a command string to execute a code file chainee.code on the boot disk:

```
program chain;
uses kbdstuff;
var s :string;
begin
  kbdbatch(1,'x*chainee~');
end.
```

| NOTE |

This form of chaining must not be used when menus are enabled because SYSTEM.MENU will be called. In order to perform chaining with menu capability, use the CHAIN procedure in the MENU unit.

7.5.5 DELAYUNIT

This unit allows scheduled delays to be executed. The DELAYUNIT operating system support routines are loaded only on systems with 128K bytes of memory. This unit consists of three procedures: (1) DELAY, which allows the suspension of a task for a number of seconds; (2) TIME_OUT_DELAY, which allows a task to suspend itself for either a number of seconds or until a semaphore has been signalled; (3) CANCEL_TIME_OUT, which is used to remove a pending time out when the semaphore parameter of TIME_OUT_DELAY has been signalled. In addition, the DELAYUNIT declares a type TIMEOBJECT, which coordinates task that are waiting for delays or time out delays. The interface for the DELAYUNIT is as follows:

```
type

 semptr =^semaphore;

 timeobject = record
             delay_sem : semptr;      {semaphore to signal to awaken}
             timed_out : ^boolean;    {set true if timed out}
             time_outhi: integer;     {time to be awakened}
             time_outlo: integer;      {time to be awakened}
             time_link : ^timeobject {points to next clocknode}
           end;

procedure time_out_delay (seconds: integer; var delaynode : timeobject;
                          var sem : semaphore; var timeout : boolean);

procedure delay (seconds : integer);

procedure cancel_time_out (var self : timeobject);
```

The procedure DELAY suspends execution of the task requesting a delay for at least its SECONDS parameter. If the system clock is running (for the clock to be running the field HAS CLOCK must be set to TRUE and the field CLOCK VALUE set to 3 or greater in the SYSTEM.MISCINFO file at system boot), the delay uses the clock to wake the task after the specified number of seconds have passed. Any other tasks are free to run while the subject task is suspended.

The procedure TIME_OUT_DELAY allows a task to suspend execution for SECONDS or until SEM is signalled, whichever comes first. The parameter DELAYNODE is set up as part of the call to TIME_OUT_DELAY.

The procedure CANCEL_TIME_OUT is called if the semaphore parameter for TIME_OUT_DELAY was signalled rather than a time out happening. If no time out happened, CANCEL_TIME_OUT must be called to remove the task from the pending time out.

If no running clock exists, the delay is simulated by a count-down loop that uses the processor (as a very low priority task) during the entire delay.

These procedures may be called by as many tasks as desired and are accurate if the system clock is running. The accuracy is determined by the tick rate of the clock and the number of competing higher priority tasks. Without a system clock, multiple delays or time outs are not accurate because a task must wait for the delays of other higher priority tasks to complete as well as wait for its own delays.

The program in Figure 7-20 is an example of two tasks that wait for character input or time out depending on user input.

```
program test;
uses  delayunit;

  (*

  procedure delay (seconds: integer);

  procedure time_out_delay (seconds: integer; var delaynode : timeobject;
                            var sem : semaphore; var timeout : boolean);

  procedure cancel_time_out (var delaynode : timeobject);

  *)

var ch          : char;
    havchar1    : semaphore;
    havchar2    : semaphore;

process inchar;
  begin
    repeat
      read (ch);
      if ch <= 'Z'
        then signal (havchar1)
        else signal (havchar2)
    until ch = '&'
  end;

process one;
  var node    : timeobject;
      timeout : boolean;
  begin
    repeat
      timeoutdelay (5,node,havchar1,timeout);
      wait (havchar1);
      if timeout
        then writeln ('timeout 1')
        else canceltimeout (node)
    until ch = '&'
  end;
```

Figure 7-2Ø. Example Program of DELAYUNIT. (Page 1 of 2)

```
process two;
  var node    : timeobject;
      timeout : boolean;
  begin
    repeat
      timeoutdelay (5,node,havchar2,timeout);
      wait (havchar2);
      if timeout
        then writeln ('timeout 2')
        else canceltimeout (node)
    until ch = '&'
  end;

begin
  ch := ' ';
  seminit (havchar1,0);
  seminit (havchar2,0);
  start (inchar,,500);
  start (one,,500);
  start (two,,500)
end.
```

Figure 7-20. Example Program of DELAYUNIT. (Page 2 of 2)

7.6 UCSD PASCAL ENHANCEMENTS

This section is a summary of the areas in which UCSD Pascal differs from Standard Pascal as well as special enhancements offered by UCSD Pascal. The Standard Pascal referenced here is defined in Pascal User Manual And Report (2nd edition) by Kathleen Jensen and Niklaus Wirth (Springer-Verlag, 1975). Many of the differences are in the areas of files and I/O. Some of the key differences from a programming standpoint are in EOF, EOLN, READ, WRITE, RESET, and REWRITE.

7.6.1 Case Statements

In Standard Pascal, if no label is equal to the value of the case statement selector, the result of the case statement is undefined (Jensen and Wirth).

In UCSD Pascal, if no label matches the value of the case selector, the next statement executed is the statement following the case statement. An example is shown in Figure 7-21. A semicolon is NOT permitted before the "END" of a case variant field declaration within a RECORD declaration. See Appendix F for revised syntax diagrams for <field list>.

```
PROGRAM FALLTHRU;
 VAR I : INTEGER;
 BEGIN (*FALLTHRU*)
  I := 25;
  CASE I OF
   Ø : WRITELN('I = Ø');
   1 : WRITELN('I = 1');
  END(*CASE*);
 WRITELN('NEITHER');
 END (*FALLTHRU*).
```

Figure 7-21. Example of Fallthrough in a Case Statement.

7.6.2 Comments

A comment is any text that appears between the symbols "(*" and "*)" or the symbols "{" and "}". Comments are ignored by the Compiler unless the first character of a comment is "$", in which case, the comment is interpreted to be a Compiler control directive. Matching symbols must be used; they may not be mixed. This feature allows a user to nest comments. For example:

```
{ XCP := XCP + 1;  (* NESTED COMMENT *) }
```

The matching symbols are a pair of different symbols. Using the same pair for nesting results in a syntax error.

7.6.3 Dynamic Memory Allocation

In Standard Pascal, DISPOSE asks that storage occupied by one particular variable be released by the system for other uses.

In UCSD Pascal, DISPOSE is not implemented. However, it can be approximated by a combined use of the intrinsics MARK and RELEASE.

Storage is allocated for variables by the standard procedure NEW in a stack-like structure called a "heap". The program in Figure 7-22 illustrates how MARK and RELEASE can be used to change the size of the heap. As NEW is used to create a new variable, the size of the heap is augmented by the size of the variable. When the variable is no longer needed, RELEASE resets the top-of-heap address that was set originally by MARK.

A series of calls to NEW between calls to MARK and RELEASE result in the immediate release of storage occupied by several variables at RELEASE time.

| NOTE |

> Because of the stack nature of the heap, memory used by a single item in the middle of the heap cannot be released. This deficiency is why MARK and RELEASE only approximate the function of DISPOSE.

Careless use of MARK and RELEASE can lead to "dangling pointers" that point to areas of memory that are no longer a part of the defined heap space.

```
PROGRAM HEAPCHNG;
 TYPE STUDENT =
         RECORD
           NAME : PACKED ARRAY [Ø..1Ø] OF CHAR;
           ID   : INTEGER
         END;
 VAR S : ^STUDENT; (* '^' MEANS POINTER*)
     HEAP : ^INTEGER;

 BEGIN (*HEAPCHNG*)
   MARK(HEAP);
   NEW(S);
   S^.NAME := 'SMITH, JOHN';
   S^.ID := 2656;
   RELEASE(HEAP);
 END (*HEAPCHNG*).
```

Figure 7-22. Using MARK and RELEASE to Change Heap Size.

7.6.4 EOF(F)

When text file F is being used as an input file from the CONSOLE device, to set EOF to True, the user must type the EOF character. The system default EOF character is control-C. (To change the default character, see Section 6.1, SETUP.)

If F is closed, EOF(F) returns True for any FILE F. If F is a file of type TEXT and EOF(F) is True, then EOLN(F) is also True. After a RESET(F), EOF(F) is False. If EOF(F) becomes True (end-of-file is reached) during a GET(F) or READ(F), the data obtained is invalid.

When a user program starts execution, the system automatically performs a RESET on the predeclared files INPUT, OUTPUT, and KEYBOARD.

The default file for EOF and EOLN is INPUT.

7.6.5 EOLN(F)

EOLN(F) is defined only if F is a text file. F is defined as a text file when the window variable F^ is of <type> CHAR. EOLN becomes True after reading the end-of-line character carriage return <cr>.

7.6.6 Files

Several aspects of file handling are described below. These enhancements bring UCSD Pascal closer to the standard definition of the language. UCSD Pascal includes untyped files that are not available to the Standard Pascal user.

| WARNING |

> READs or WRITEs to files of types other than TEXT or FILE OF CHAR may not be done. Instead, a GET or PUT must be done.

INTERACTIVE FILES

> The standard predeclared files INPUT and OUTPUT are always defined as type INTERACTIVE and behave exactly as do files of type TEXT. All files other than INTERACTIVE operate exactly as described in Jensen and Wirth, including the functioning of EOF(F), EOLN(F) and RESET(F). For more details concerning files of type INTERACTIVE, see Sections 7.6.11 (READ and READLN) and 7.6.12 (RESET).

UNTYPED FILES

Untyped files are unique to UCSD Pascal. An untyped file can be thought of as a file without a window variable F^ to which all I/O must be accomplished (using BLOCKREAD and BLOCKWRITE). Any number of blocks can be transferred using either BLOCKREAD or BLOCKWRITE. These functions return the actual number of blocks read or written. When untyped files are used, Compile option {$I-} should be specified, thus requiring that the function IORESULT and the number of blocks transferred be explicitly checked after each BLOCKREAD or BLOCK WRITE to detect any I/O errors. An example of a program that uses untyped files is shown in Figure 7-23.

```
(*$I-*)
PROGRAM FILEXAMP;
 VAR S,D : FILE;
     BUF : PACKED ARRAY[Ø..511] OF CHAR;
     BLKN, BLKSTRAN : INTEGER;
     IOERR : BOOLEAN;
 BEGIN (*FILEXAMP*);
  IOERR := FALSE;
  RESET(S,'FROM.DATA');
  REWRITE(D,'TO');
  BLKN := Ø;
  BLKSTRAN := BLOCKREAD(S,BUF,1,BLKN);
  WHILE (NOT EOF(S)) AND (IORESULT = Ø)
          AND (NOT IOERR) AND (BLKSTRAN=1) DO
   BEGIN
    BLKSTRAN := BLOCKWRITE(D,BUF,1,BLKN);
    IOERR := ((BLKSTRAN < 1) OR (IORESULT <> Ø));
    BLKN := BLKN + 1;
    BLKSTRAN := BLOCKREAD(S,BUF,1,BLKN);
   END (*WHILE*);
  CLOSE(D,LOCK);
 END (*FILEXAMP*).
```

Figure 7-23. Example of Using Untyped Files.

RANDOM ACCESS OF FILES

Individual records in a file can be accessed randomly by the intrinsic SEEK. The two parameters for SEEK are the file identifier and an integer giving the record number to which the window should be moved. The first record of a structured file has the number Ø. SEEK always sets EOF and EOLN to False. The subsequent GET or PUT sets these conditions as appropriate. Attempts to PUT records beyond the physical end-of-file sets EOF to True.

7.6.7 GOTO and EXIT Statements

The GOTO statement may not branch to a label that is not within the same block as the statement. This limitation is not imposed on the GOTO statement in Standard Pascal. Because of this limitation, the examples on pages 31-32 of Jensen and Wirth do not apply.

|NOTE|

The GOTO statement receives a syntax error during compilation unless the {$G+} option is enabled.

EXIT is a UCSD extension statement. Its only parameter is the identifier of the procedure to be exited. The EXIT statement was created because of the occasional need for a means to abort a complicated, and possibly deeply nested, series of procedure calls on encountering an error. EXIT(program) terminates execution of a program.

| NOTE |

The use of an EXIT statement to exit a function can result in the function returning undefined values if no assignment is made to the function identifier prior to the execution of the EXIT statement.

If the identifier in the EXIT statement is that of a recursive procedure, the most recent invocation of that procedure is EXITed. Upon EXIT, an implicit CLOSE(F) is done on local files that were opened during execution of the procedure being EXITed. An example of using EXIT is shown in Figure 7-24.

```
PROGRAM EXITTEST;
VAR S : STRING;
  I : INTEGER;

  PROCEDURE CALL; FORWARD;

  PROCEDURE PRINT;
   BEGIN (*PRINT*)
    WRITELN('-->');
    READLN(S);
    WRITELN(S);
    IF S[1] = '*' THEN EXIT(CALL);
    WRITELN('LEAVE PRINT');
   END (*PRINT*);

   PROCEDURE CALL;
    BEGIN (*CALL*)
     PRINT;
     WRITELN('LEAVE CALL');
    END (*CALL*);

   PROCEDURE COUNT;
    BEGIN (*COUNT*)
     IF I <= 1Ø THEN CALL;
     WRITELN('LEAVE COUNT');
    END (*COUNT*);

   BEGIN (*EXITTEST*)
     I := Ø;
     WHILE NOT EOF DO
      BEGIN
       I := I+1;
       COUNT;
       WRITELN;
      END (*WHILE*);
   END (*EXITTEST*).
```

Figure 7-24. Example of Using the EXIT Statement.

7.6.8 Packed Variables

Packed arrays and records, using packed variables as parameters, are described below. These packed arrays and records do NOT use the procedures PACK and UNPACK.

PACKED ARRAYS

The UCSD Pascal Compiler packs arrays if the ARRAY declaration is preceded by the word PACKED. For example:

```
ARRAY[Ø..9] OF CHAR;
PACKED ARRAY[Ø..9] OF CHAR;
```

The array in the first declaration occupies ten 16-bit words of memory, with each element occupying one word. The array in the second declaration is packed into a total of five words, because each 16-bit word contains two 8-bit characters. Thus, each element is eight bits long.

Examples of packed arrays that are not of type CHAR are given in Figure 7-25.

```
PROGRAM PACKTST;
 VAR A: PACKED ARRAY [Ø..9] OF Ø..255;    {5 words of memory}
                                          {allocated.}
     B: PACKED ARRAY [Ø..15] OF BOOLEAN;  {1 word}
     C: PACKED RECORD
          D: BOOLEAN;          {5 words for VAR C}
          CASE E: BOOLEAN OF
          TRUE: (F: INTEGER);
          FALSE: (G: PACKED ARRAY [Ø..7] OF CHAR)
          END;
 BEGIN
 END.
```

Figure 7-25. Examples of Packed Arrays and Records.

Because of the recursive nature of the Compiler, the following two declarations are not equivalent:

```
PACKED ARRAY[Ø..9] OF ARRAY[Ø..3] OF CHAR;
PACKED ARRAY[Ø..9,Ø..3] OF CHAR;
```

In the first declaration, the second occurrence of ARRAY causes packing in the Compiler to be turned off, giving an unpacked array of 40 words. The array in the second declaration occupies a total of 20 words because ARRAY appears only once. If the second occurrence of ARRAY in the first declaration had also been preceded by PACKED, the two declarations would have been equivalent.

An array will be packed only if the final type of array is scalar, subrange, or a set that can be represented in eight bits or less or if the final type is BOOLEAN or CHAR. No packing is done if the array cannot be expressed in a field of eight bits.

No packing occurs across word boundaries. If the type of element to be packed requires a number of bits that does not divide evenly by 16, unused bits are at the high end of each of the words that comprise the array.

| NOTE |

> Assigning a string constant to an unpacked ARRAY OF CHAR is illegal, although it may be assigned to a PACKED ARRAY OF CHAR. Also, comparisons between an ARRAY OF CHAR and a string constant are illegal. These restrictions are because of size differences.

A PACKED ARRAY OF CHAR may be output with a single WRITE statement and may be initialized by using the intrinsics FILLCHAR and SIZEOF.

PACKED RECORDS

As with arrays, the Compiler packs records if the RECORD declaration is preceded by PACKED. In the example below, the entire record is packed into one 16-bit word.

```
VAR A:  PACKED RECORD
         Q,R,S: 0..31;
         B: BOOLEAN
        END;
```

The variables Q, R, and S each take up five bits. The Boolean variable is allocated to the sixteenth bit.

PACKED RECORDS may contain fields that also are packed, either arrays or records. But PACKED must occur before every occurrence of RECORD to retain packed qualities throughout all fields of the record.

A case variant may only be used as the last field of a packed or unpacked record. The amount of space allocated to it is the size of the largest variant among the cases.

PACK AND UNPACK

UCSD Pascal does NOT support the standard procedures PACK and UNPACK. (Jensen and Wirth, 1Ø6).

7.6.9 Parametric Procedures and Functions

UCSD Pascal does NOT support the use of procedures and functions as formal parameters in the parameter list of a procedure or function.

7.6.1Ø Program Headings

A list of file parameters may follow the program identifier. However, they are ignored by the Compiler and have no effect on the program being compiled. Any file declarations other than the three predeclared files (INPUT, OUTPUT, and KEYBOARD) of type INTERACTIVE must be declared along with the other VAR declarations for the program.

7.6.11 READ and READLN

In Standard Pascal, the procedure READ requires that the window variable F^ be loaded with the first character of the file when the file is opened. If effect, the statement READ(F,CH) would be equivalent to:

```
CH: =F^;
GET(F);
```

To be responsive to the demands of an interactive programming environment, UCSD Pascal defines the additional file type INTERACTIVE. Declaring a file to be of type INTERACTIVE is equivalent to declaring it to be type TEXT, except that the definition of READ(F,CH) is reversed:

```
GET(F);
CH: =F^;
```

The standard definition of the procedure READ requires that the process of opening a file include loading the window variable F^ with the first character of the file. In an interactive environment, it is inconvenient to require a user to type a character of the input file when it is open to avoid the program "hanging" when it is first opened. To overcome this, UCSD Pascal has reversed the order. This difference affects the way in which EOLN must be used when reading from a text file of the type INTERACTIVE. EOLN only becomes true after reading the end-of-line character, a <return>. The character returned as a result of the READ is a blank.

Three predeclared text files (INPUT, OUTPUT, and KEYBOARD) of type INTERACTIVE are opened automatically for a user program. The file INPUT defaults to the console device. The statement READ(INPUT,CH), where CH is a character variable, echoes the character typed from the console back to the console. WRITE statements to the file OUTPUT cause the output to appear on the console, by default. The file KEYBOARD is the nonechoing equivalent to INPUT. For example, the following two statements are equivalent to READ(INPUT,CH);

```
READ (KEYBOARD, CH);

WRITE (OUTPUT, CH);
```

7.6.12 RESET(F)

In Standard Pascal, the procedure RESET resets the file window to the beginning of file F. The next GET(F) or PUT(F) affects record Ø of the file. Also, the window variable F^ is loaded with the first record of the file.

In UCSD Pascal, the standard conventions hold true unless the file is of type INTERACTIVE. In that case, the window variable is NOT loaded. Thus, the UCSD equivalent of the Standard RESET(F) for a file of type interactive is the two-statement sequence:

```
RESET(F);
GET(F);
```

UCSD Pascal also provides an alternative form of opening a pre-existing file. In it, RESET has two parameters; the file identifier followed by either a string constant or variable, whichever corresponds to the directory file name of the file being reopened.

Examples:

```
S := 'NAME.TEXT';
RESET(F,S);  {Opens NAME.TEXT on the prefixed volume}

RESET(F,'REMOTE:');  {Allows input to and output from}
                     {REMOTE, (serial port B)}
WRITELN(F,'This is the remote terminal');
```

7.6.13 REWRITE(F,S)

REWRITE opens and creates a new file. It has two parameters: the file identifier followed by either a string constant or variable, giving the title of the file being created. The file name may include a block size specification. (See 7.2.2, REWRITE.)

7.6.14 Segment Procedures

The SEGMENT PROCEDURE is a UCSD extension to Pascal. With it, the programmer can segment a large program so that the entire program need not be in memory at once. For further information, see Section 7.3, Segments.

7.6.15 Sets

All of the Standard Pascal constructs for sets are supported by UCSD Pascal. (See p. 50-51 of Jensen and Wirth.) Sets of enumeration values are limited to positive integers only. Also, a set is limited to 255 words and 4080 elements. Comparisons and operations are allowed only between sets that are either of the same base type or subranges of the same underlying type. Examples are shown in Figure 7-26.

```
PROGRAM SETST;
 VAR SET1: SET OF 0..49;
     SET2: SET OF 0..99;

 BEGIN (*SETST*)
   SET1 := [0,5,10];
   SET2 := [10,20,30];
   IF SET1 = SET2 THEN
      WRITELN('THEY ARE EQUAL')
   ELSE
      WRITELN('THEY ARE NOT EQUAL');
 END(*SETST*).
```

Sets of different underlying types cannot be compared:

```
PROGRAM SETCOMP;
 TYPE INGREDIENTS = (FLOUR,SUGAR,EGGS,SALT);

 VAR I: SET OF INGREDIENTS;
     N: SET OF 0..49;

 BEGIN (*SETCOMP*)
   I := [FLOUR];
   N := [1,2,3,4,5];
   IF I = N THEN    <----- SYNTAX ERROR WILL OCCUR HERE
   WRITELN('EQUAL');
 END (*SETCOMP*).
```

Figure 7-26. Examples of Set Comparisons.

7.6.16 Strings

STRING variables are unique to UCSD Pascal. Essentially, they are PACKED ARRAYs of CHAR with a dynamic LENGTH attribute, the value of which is returned by the string intrinsic LENGTH. The default maximum length of a string variable is 80 characters. This value can be overridden in the declaration of a string by appending the desired length within [] after the type identifier STRING. For further information and examples, see Section 7.2.3, String Intrinsics.

A string variable has an absolute maximum length of 255 characters. Assignment to string variables can be performed using the assignment statement, using UCSD string intrinsics, or using a READ statement. For example:

```
TITLE:=' THIS IS MY STRING ';
READLN(MYSTRING);
NAME:= COPY(MYSTRING,1,21);
```

The individual characters within a string are indexed from 1 to the length of the string. A string variable may not be indexed beyond its current dynamic length; otherwise, a run-time error is generated.

String variables may be compared (=, <>, >, <, >=, <=) to other string variables, no matter what the current dynamic length of either. If the lengths of two strings being compared are unequal, the shorter string is extended to the length of the longer by appending blanks. Comparison is based on the ASCII collating sequence.

A common use of string variables in UCSD Pascal is reading file names from the console device. When a string variable is used as a parameter to READ or READLN, all characters up to the end-of-line character (carriage return) in the source file are assigned to the string variable. In reading string variables, the single statement READLN(S1,S2) is equivalent to the two-statement sequence:

```
READ(S1);
READLN(S2);
```

7.6.17 WRITE and WRITELN

The procedures WRITE and WRITELN follow the conventions of Standard Pascal except when applied to a variable of type BOOLEAN. UCSD Pascal does not support the output of the words TRUE or FALSE when writing out the value of a Boolean variable. In order to write out Boolean values, the ORD function must be used.

For writing variables of type STRING, see Section 7.2.3, String Intrinsics. When a string variable is written without specifying a field width, the actual number of characters written is equal to the dynamic length of the string. If the field width specified is longer than the dynamic length, leading blanks are inserted. If the field width is smaller, excess characters are truncated on the right.

7.6.18 Implementation Size Limits

The maximum size limitations of UCSD Pascal are shown below.

- Maximum number of bytes of object code in a procedure or function is 1200. Maximum number of words for local variables in a procedure or function is 32676.
- Maximum number of characters in a string variable is 255.
- Maximum number of elements in a set is 255 * 16 = 4080.
- Maximum number of user segments available is 128 (numbered 128..255). Nine system segments (1, 8..15) are also available for user programs.
- Maximum number of procedures or functions within a segment is 255.
- Maximum number of bytes in a segment is 65535.

7.6.19 Extended Comparisons

UCSD Pascal permits = and <> comparisons of any array or record structure.

7.6.2Ø Data Types for Concurrency

Three data types for concurrency are available. See Section 7.7.4 Concurrency Primatives and Interrupts for more information.

PROCESS Type

A process declaration creates a task that may run concurrently with other tasks in the system. A task is invoked by the START statement. The process declaration is syntactically similar to the procedure declaration with the word "procedure" being replaced by "process".

Example:

```
PROCESS CONTROLLER(TEMP, PRESSURE: REAL);
  VAR T : INTEGER;
BEGIN
  ...
END;
```

PROCESSID Type

A processid is a pointer to a Task Information Block (TIB). See Section 7.7.3 Registers and Operating System Tables for a description of the TIB. The START statement has as one of its parameters a variable of type processid.

Example:

```
VAR PID : PROCESSID;
```

SEMAPHORE Type

A semaphore is a synchronization primitive that provides synchronization between tasks and detection of hardware interrupts with the functions SIGNAL, WAIT, ATTACH, and SEMINIT. The internal structure of a semaphore is described in Section 7.7.4 Concurrenty Primitives and Interrupts.

Example:

```
VAR SEM : SEMAPHORE;
```

7.6.21 Programming Examples

This section gives programming examples for three subject areas: (1) absolute memory locations, (2) I/O drivers, and (3) directory access.

Absolute Memory Locations

Referencing absolute memory locations on the 1600 Series Computer Systems is performed through Pascal variant records. A variant record specifies that two different variables with possibly different types may occupy the same memory location.

| NOTE |

> Absolute addressing is very powerful, and the system operating tables or code can be corrupted by improper usage. Because the 1600 series computers have mapped I/O, the I/O control registers and drivers can be addressed and, thus, could be easily corrupted.

Figure 7-27 is a program that accesses an absolute memory address interactively.

```
PROGRAM EXAMINE;

TYPE MEMREC = RECORD
                MEMCELL : INTEGER
              END;


VAR MEMVARIANT : RECORD CASE BOOLEAN OF
                   TRUE  : (MEMADD : INTEGER);
                   FALSE : (MEMCONTS : ^MEMREC);
                 END;


    I : INTEGER;


BEGIN
  WRITE (' ENTER ABSOLUTE ADDRESS ');
  READLN (i);
  MEMVARIANT.MEMADD  :=I;
  WRITELN (' CONTENTS OF ',i,' = ', MEMVARIANT.MEMCONTS^.MEMCELL);
END.
```

{If an address of a MicroEngine I/O port were entered, the program would return the contents of the I/O port register.}

Figure 7-27. Program to Access Absolute Memory Address.

I/O Drivers

The I/O controller registers can be referenced by absolute memory accessing. The three programs given reference noninterrupt and interrupt I/O drivers.

Figure 7-28 shows a program that outputs characters to the console by using variant records. The program writes to the serial port using a declared procedure unitwrite.

| NOTE |

This form of I/O driver may only be run on noninterrupt operating systems (releases prior to HØ).

```
PROGRAM SERIALTEST;

TYPE
  STATCMDEC = RECORD CASE BOOLEAN OF
    TRUE : (COMMAND : INTEGER);
    FALSE : (STATUS : PACKED ARRAY[Ø,,7] OF BOOLEAN);
  END; (* For devices that use same reg for stat and cmd*)

  SERIALREC = RECORD
              SERDATA : INTEGER;
              STATSYNDLE : STATCMDREC;
              CONTROL2 : INTEGER;
              CONTROL1 : INTEGER
              FILLER : ARRAY[Ø..3] OF INTEGER;
              SWITCH : STATCMDREC;
            END;
  VAR
     SERIALTRIX : RECORD CASE INTEGER OF
              Ø : (DEVADD : INTEGER) ;
              1 : (SERIAL : ^SERIALREC);
            END;

     PROCEDURE SUNITWRITE (CH: CHAR);

     BEGIN
       WITH SERIALTRIX DO
         BEGIN
           DEVADD :=-1ØØ8; (* FC1Ø *)
           WITH SERIAL^ DO
             BEGIN
               CONTROL1 :=135; (*87 HEX *)
               CONTROL1 :=1;   (* Ø1 *)
               REPEAT
                 UNTIL STATSYNDLE.STATUS[Ø];
               SERDATA  := ORD(CH);
             END;
         END;
     END;

     BEGIN
       SUNITWRITE ('h'); SUNITWRITE ( 'i');
     END.
```

Figure 7-28. Noninterrupt I/O Driver-Referenced by Absolute Memory Address.

Figures 7-29 and 7-30 show examples of programs that are interrupt driven I/O drivers by use of variant records. Figure 7-29 is a program for CONSOLE:, and Figure 7-30, for REMOTE:.

| NOTE |

This form of I/O driver may only be run on interrupt operating systems (release H0 and greater).

```
{ This program is an example of a serial port A receiver interrupt driver }

PROGRAM test;
 type
     memtrix= record case boolean of
                true: (addr: integer);
                false:(loc: ^integer);
              end;
     statcmdrec = record case boolean of
                    true : (command : integer);
                    false : (status : packed array[0..7] of boolean);
                  end;  { for devices that use same reg for stat and cmd }
     whole = 0..maxint;

     serialrec = record
                   data : integer;
                   statsyndle : statcmdrec;
                   control2 : integer;
                   control1 : integer;
                   filler : integer;
                   switch : statcmdrec;
                   filler2 : array [0..1] of integer;
                   switch2 : statcmdrec;
                 end;
```

Figure 7-29. Interrupt I/O Driver - Referenced by Absolute Memory Address. (CONSOLE:) (Page 1 of 2)

```
var
    i: integer;
    lstatus: statcmdrec;
    serialtrix : record case boolean of
                    true : (sdevadd : integer);
                    false : (serial : ^serialrec);
                  end;
    lmem: memtrix;
    serAintconts: memtrix;
    saveaint : integer;
    sersem : semaphore;

PROCESS serread;
   BEGIN
      REPEAT
        wait(sersem);
        i := serialtrix.serial^.data;
        lmem.loc^ := 1;
        write(chr(i));
            { Some terminals set the high order bit so add 128 }
      UNTIL (i = ord('q')) or (i = ord('Q')) or (i = 209) or (i = 241);
      serAintconts.loc^ := saveaint; { Restore OS driver }
   END; {serread}

BEGIN
   lmem.addr := -952; { Re-enable interrupt address }
   serialtrix.sdevadd := -1008; { Serial port a device address }
   with serialtrix,serial^ do
     begin control1 := 133; control2 := 1; end;
   serAintconts.addr := 36;
   saveaint := serAintconts.loc^; { Save OS driver interrupt semaphore }
   seminit(sersem,0);  attach(sersem, 36);
   start(serread);
END.
```

Figure 7-29. Interrupt I/O Driver - Referenced by Absolute Memory Address. (CONSOLE:) (Page 2 of 2)

```
{ This program is an example of interrupt driven I/O drivers for the
    remote port.  These drivers handle serial input,output, and carrier
    detect interrupts.  In addition, an output driver for the console
    is illustrated.
}

PROGRAM test;
 type
     memtrix= record case boolean of
                true: (addr: integer);
                false:(loc: ^integer);
              end;
     statcmdrec = record case boolean of
                    true : (command : integer);
                    false : (status : packed array[Ø..7] of boolean);
                  end;  { for devices that use same reg for stat and cmd }
     serialrec = record
                   data : integer;
                   statsyndle : statcmdrec;
                   control2 : integer;
                   control1 : integer;
                   filler : integer;
                   switch : statcmdrec;
                   filler2 : array [Ø..1] of integer;
                   switch2 : statcmdrec;
                 end;

    var
       j,i: integer;
       ch: char;
       lstatus: statcmdrec;
       rserialtrix : record case boolean of
                       true : (sdevadd : integer);
                       false : (serial : ^serialrec);
                     end;
       cserialtrix : record case boolean of
                       true : (sdevadd : integer);
                       false : (serial : ^serialrec);
                     end;
      enabletrix: memtrix;
      cwritesem,rwritesem,xcptsem,rreadsem : semaphore;
```

Figure 7-3Ø. Interrupt I/O Driver - Referenced by Absolute Memory Address.
(REMOTE:) (Page 1 of 3)

```
procedure rserwrite(charo: integer);
begin
  rserialtrix.serial^.data:= charo;
  rserialtrix.serial^.controll:= 135;
  wait (rwritesem);
  repeat until rserialtrix.serial^.statsyndle.status[5];
  rserialtrix.serial^.controll:= 133;
  enabletrix.loc^ := 1;
end;

procedure cserwrite(charo: integer);
begin
  cserialtrix.serial^.data:= charo;
  cserialtrix.serial^.controll:= 135;
  wait (cwritesem);
  cserialtrix.serial^.controll:= 133;
  enabletrix.loc^ := 1;
end;

PROCESS serxcpt;
   BEGIN
      REPEAT
         wait(xcptsem);
         lstatus := rserialtrix.serial^.statsyndle;
         enabletrix.loc^ := 1;
      UNTIL false;
   END; {serxcpt}
```

Figure 7-3Ø. Interrupt I/O Driver - Referenced by Absolute Memory Address. (REMOTE:) (Page 2 of 3)

```
PROCESS serread;
   BEGIN
      REPEAT
         wait(rreadsem);
         rserialtrix.serial^.controll := 129;
         i := rserialtrix.serial^.data;
         enabletrix.loc^ := 1;
         cserwrite(i);
         rserialtrix.serial^.controll := 133;
      UNTIL false;
   END; {serread}

BEGIN
   enabletrix.addr := -952;
   rserialtrix.sdevadd := -992; cserialtrix.sdevadd := -1ØØ8;
   seminit(rreadsem,Ø);  seminit(xcptsem,Ø); seminit(rwritesem,Ø);
   seminit(cwritesem,Ø);
   attach(rreadsem, 44); attach(xcptsem,52); attach(rwritesem,4Ø);
   attach(cwritesem,48);
   start(serread); start(serxcpt);
   for j := 1 to 1Ø do
     begin
       ch := 'a';
       for i := 1 to 26 do
         begin
           rserwrite(ord(ch));
           ch := succ(ch);
         end;
     end;
END.
```

Figure 7-3Ø. Interrupt I/O Driver - Referenced by Absolute Memory Address. (REMOTE:) (Page 3 of 3)

Directory Access

A diskette is composed of granules called blocks. Each block contains 512 bytes. A single-sided, single-density diskette contains 494 blocks numbered from Ø - 493. A double-sided, double-density diskette contains 1,976 blocks.

The directory for a diskette resides on block numbers 2-5 (occupies 4 disk blocks). If a duplicate directory exists, it resides on blocks 6-9. Among other things, the directory contains the name of the diskette, the name of each file on the diskette, information concerning the starting and ending block for each file, and the date of creation of each file.

Figure 7-31 shows the Pascal declaration for a directory. This declaration is identical to the one in the operating system globals. In addition, a Pascal program fragment that reads the date stored in the directory is illustrated.

```
DATEREC = PACKED RECORD
           MONTH: Ø..12;
           DAY: Ø..31;
           YEAR: Ø..1ØØ;
         END;
DIRENTRY = RECORD
             DFIRSTBLK: INTEGER,  {First physical disk addr}
             DLASTBLK: INTEGER,  {Points at block following}
             CASE DFKIND:  FILEKIND OF
               SECUREDIR,
               UNTYPEFILE:  {only in dir [Ø] ... volume info}
                 (DVID: VID;
                 DEOVBLK: INTEGER,          {Lastblk of volume}
                 DNUMFILES: DIRRANGE;       {Num files in dir}
                 DLOADTIME: INTEGER;        {Time of last access}
                 DLASTBOOT: DATEREC);       {Most recent date setting}
           XDSKFILE,CODEFILE,TEXTFILE,INFOFILE,
           DATAFILE,GRAFFILE,FOTOFILE:
                 (DTID: TID;
                 DLASTBYTE: 1..FBLKSIZE;    {Num bytes in last block}
                    DACCESS:DATEREC)        {Last modification date}
           END (Direntry} ;

DIRP = ^DIRECTORY;

DIRECTORY = ARRAY [DIRRANGE] OF DIRENTRY;


{The following program fragment reads the directory from disk drive #4}

   VAR GDIRP: DIRP;

   BEGIN
     NEW (GDIRP);
     UNITREAD (4, GDIRP^, SIZEOF (DIRECTORY),2);

{After this read from disk of the directory, the fields in the directory
 may be examined.  For example, to access the date on the diskette:}

 WITH GDIRP^[Ø].DLASTBOOT DO

   WRITELN ('TODAY IS', MONTH, '/', DAY '/',YEAR);
```

Figure 7-31. Directory Access.

APPENDIX A. COMMAND SUMMARIES

A.1 OUTER LEVEL

C(omp	Invokes the system Compiler.
E(dit	Invokes the system Editor (Screen-Oriented Editor or Yet Another Line-Oriented Editor).
eX(ecute	Executes a code file.
F(iler	Invokes the File Handler.
L(ink	Invokes the Linker.
R(un	Executes the code file associated with the current work file. If none exists, the Compiler is automatically called, followed by the Linker, if necessary, before execution.
D(ebug	Invokes the Debugger. If no code file exists, the Compiler is automatically called, followed by the Linker, if necessary.
I(nitialize	Reinitializes the system.
H(alt	Halts the machine. Reboot is required.
A(da	Invokes the MicroAda(TM) compiler.
U(ser restart	Reexecutes program or option last used.

A.2 SCREEN-ORIENTED EDITOR

<down-arrow>	moves <repeat-factor> lines down
<up-arrow>	" " lines up
<right-arrow>	" " spaces right
<left-arrow>	" " spaces left
<space>	" " spaces in direction
<back-space>	" " spaces left
<tab>	moves <repeat-factor> tab positions in direction
<return>	moves to the beginning of line <repeat-factor> lines in direction.
"<" "," "-"	changes direction to backward
">" "." "+"	changes direction to forward
"="	moves to the beginning of data that were just found/replaced/inserted/adjusted

A(djust

Adjusts the indentation of the line on which the cursor is located. Use the arrow keys to move. Moving up (or down) adjusts line above (or below) by same amount of adjustment as current line. Repeat factors are valid.

C(opy

Copies information that was last inserted/deleted/zapped into the file at the position of the cursor.

D(elete

Deletes data using the starting position of the cursor as the anchor. Use any moving commands to move the cursor. <etx> accepts deletion; everything between the cursor and the anchor is deleted.

F(ind

Operates in L(iteral or T(oken mode. Finds the <targ> string. Repeat factors are valid; direction is applied. "S" uses the same string just previously used.

I(nsert

Inserts text. Can use <backspace> and <del> to reject part of your insertion.

J(ump

Jumps to the beginning, end, or previously set marker.

M(argin

Adjusts anything between two blank lines to the margins that are set. Command characters protect text from being margined. Invalidates the copy buffer.

P(age

Moves the cursor one page in direction. Repeat factors are valid; direction is applied.

Q(uit

Leaves the Editor. Options are to U)pdate, E)xit, W)rite, or R)eturn.

R(eplace

Operates in L(iteral or T(oken mode. Replaces the <targ> string with the <subs> string. V(erify option asks you to verify before it replaces. "S" option uses the same string as just previously used. Repeat factors replace the target several times. Direction is valid.

S(et

Sets M(arkers by assigning a string name to them. Sets E(nvironment for A(uto-indent, F(illing, margins, T(oken, and C(ommand characters.

V(erify

Redisplays the screen with the cursor centered.

eX(change

Exchanges the current text for the text typed while in this mode. Each line must be done separately. <back-space> causes the original character to reappear.

Z(ap

Treats the starting position of the last thing found/replaced/inserted /adjusted as an anchor and deletes everything between the anchor and the current cursor position.

<repeat-factor>

Is any number typed before a command. Typing a / implies an infinite number.

A.3 YALOE (Yet Another Line-Oriented Editor)

n - an argument m - macro number

nA: Advance the cursor to the beginning of the nth line from the current position.
B: Go to the beginning of the file.
nC: Change by deleting n characters and inserting the following text. Terminate text with <esc>.
nD: Delete n characters.
E: Erase the screen.
nF: Find the nth occurrence from the current cursor position of the following string. Terminate target string with <esc>.
nG: Get - add FIND text except leave GET
H: - invalid command -
I: Insert the following text. Terminate text with <esc>.
nJ: Jump cursor n characters.
nK: Kill n lines of text. If current cursor position is not at the start of the line, the first part of the line remains.
nL: List n lines of text.
mM: Define macro number m.
nNm: Perform macro number m, n times.
nO: On, off toggle. If on, n lines of text are displayed above and below the cursor each time the cursor is moved. If the cursor is in the middle of a line, then the line is split into two parts. The default is whatever fills the screen. Type 0 to turn off.
P: - invalid command -
Q: Quit this session, followed by:
    U:(pdate Write out a new SYSTEM.WRK.TEXT
    E:(scape Escape from session
    R:(eturn Return to editor
R: Read this file into buffer (insert at cursor);
    'R' must be followed by <file name> <esc>;

| WARNING |

If the file does not fit into the buffer, the content of the buffer becomes undefined!

nS: Put the next n lines of text from the cursor position into the Save Buffer.
T: - invalid command -
U: Insert (Unsave) the contents of the Save Buffer into the text at the cursor; does not destroy the Save Buffer.
V: Verify: display the current line
W: Write this file (from start of buffer); 'W' must be followed by <file name> <esc>
nX: Delete n lines of text, and insert the following text; terminate with <esc>
Y: - invalid command -
Z: - invalid command -

A.4 FILE HANDLER

B(ad-blks	Scans the disk and detects bad blocks, listing the number of each.
C(hange	Changes file or volume name.
D(ate	Lists current system date and enables user to change date; format is dd-mmm-yy.
E(xt-dir	Lists the directory in more detail than the L(dir command.
G(et	Loads the designated file into the work file.
K(runch	Moves the files on the specified volume so that unused blocks are combined at the end of the disk; disk files only.
L(dir	Lists a disk directory, or subset of one, to the volume and file specified; default is CONSOLE:.
M(ake	Creates a directory entry with the specified filename.
N(ew	Clears the work file.
P(refix	Changes the current default volume to the volume specified.
Q(uit	Returns control to the Outer Level of commands.

R(em Removes file entries from the directory.

S(ave Saves the work file under the specified file name.

T(rans Copies (transfers) the specified file to the specified destination volume.

V(ols Lists the volumes currently on-line along with their corresponding device numbers.

W(hat Identifies the file name and state (saved or not) of the work file.

X(amine Attempts to recover bad blocks physically; a bad-block scan should be done first.

Z(ero Reformats the specified volume and makes the old directory irretrievable.

A.5 PASCAL COMPILER

G Affects whether the compiler allows the use of the Pascal GOTO statement in the program. Default is '-', not allowed.

I When followed by a '+', causes the Compiler to generate code after any I/O statement to check for successful completion of I/O. This setting is the default.

When followed by a '-', inhibits I/O checking.

When followed by a file name, includes another source file into the compilation.

L Causes the Compiler to generate a listing of the source program on a specified file. If a '+' is used, the default file is *SYSTEM.LIST.TEXT. The default is '-', which produces no listing.

Q "Quiet compile" option is used to suppress output to the CONSOLE of the procedure names and line numbers during compilation. The default is set to the current value of SYSCOM^.MISCINFO.SLOW-TERM.

R Affects whether the Compiler inserts code for checking on array subscripts and assignments to variables of subrange types. The default is '+'; therefore, code for checking is inserted.

S Causes the Compiler to operate in swapping mode so that only one of the two main parts of the Compiler (declarations processor or statement handler) is in main memory at one time, freeing about 2500 words for symbol table storage. The default is '-', which means that no swapping occurs.

U Determine whether the compilation is of a user program or a system program. The default is '+', which means a user program.

When followed by a file name, U names the library file to be used.

APPENDIX B: TABLES

B.1 RUN-TIME ERRORS

Version 3.0

0	System error	FATAL
1	Invalid index, value out of range	
2	No segment, bad code file	
3	Procedure not present at exit time	
4	Stack overflow	
5	Integer overflow	
6	Divide by zero	
7	Invalid memory reference <bus timed out>	
8	User Break	
9	System I/O	FATAL
10	User I/O	
11	Unimplemented instruction	
12	Floating Point math error	
13	String too long	
14	Halt, Breakpoint (without debugger in core)	

All fatal errors either cause the system to rebootstrap, or if the error was totally lethal to the system, the user must reboot by pressing the reset buttton. All errors cause the system to reinitialize itself (call system procedure INITIALIZE).

B.2 I/O RESULTS

Version 3.0

0	No error
1	Bad Block, Parity error (CRC)
2	Bad Unit Number
3	Bad Mode, Illegal operation
4	Undefined hardware error
5	Lost unit, Unit is no longer on-line
6	Lost file, File is no longer in directory
7	Bad title, Illegal file name
8	No room, insufficient space
9	No unit, No such volume on line
10	No file, No such file on volume
11	Duplicate file
12	Not closed, attempt to open an open file
13	Not open, attempt to access a closed file
14	Bad format, error in reading real or integer
15	Volume write protected

B.3 PASCAL COMPILER SYNTAX ERRORS

Version 3.0

1: Error in simple type
2: Identifier expected
3: 'PROGRAM' expected
4: ')' expected
5: ':' expected
6: Illegal symbol
7: Error in parameter list
8: 'OF' expected
9: '(' expected
10: Error in type
11: '[' expected
12: ']' expected
13: 'END' expected
14: ';' expected
15: Integer expected
16: '=' expected
17: 'BEGIN' expected
18: Error in declaration part
19: Error in <field-list>
20: '.' expected
21: '*' expected
22: 'Interface' expected
23: 'Implementation' expected
24: 'Unit' expected

50: Error in constant
51: ':=' expected
52: 'THEN' expected
53: 'UNTIL' expected
54: 'DO' expected
55: 'TO' or 'DOWNTO' expected in for statement
56: 'IF' expected
57: 'FILE' expected
58: Error in <factor> (bad expression)
59: Error in variable
60: Must be semaphore
61: Must be processid

101: Identifier declared twice
102: Low bound exceeds high bound
103: Identifier is not of the appropriate class
104: Undeclared identifier
105: Sign not allowed
106: Number expected
107: Incompatible subrange types
108: File not allowed here
109: Type must not be real
110: <tagfield> type must be scalar or subrange
111: Incompatible with <tagfield> part
112: Index type must not be real
113: Index type must be a scalar or a subrange
114: Base type must not be real
115: Base type must be a scalar or a subrange
116: Error in type of standard procedure parameter
117: Unsatisfied forward reference
118: Forward reference type identifier in variable declaration
119: Respecified params not OK for a forward declared procedure
120: Function result type must be scalar, subrange or pointer
121: File value parameter not allowed
122: A result type of the forward declared function can't be respecified
123: Missing result type in function declaration

124: F-format for reals only
125: Error in type of standard function parameter
126: Number of parameters does not agree with declaration
127: Illegal parameter substitution
128: Result type does not agree with declaration
129: Type conflict of operands
130: Expression is not of set type
131: Tests on equality allowed only
132: Strict inclusion not allowed
133: File comparison not allowed
134: Illegal type of operand(s)
135: Type of operand must be Boolean
136: Set element type must be scalar or subrange
137: Set element types must be compatible
138: Type of variable is not array
139: Index type is not compatible with the declaration
140: Type of variable is not record
141: Type of variable must be file or pointer
142: Illegal parameter substitution
143: Illegal type of loop control variable
144: Illegal type of expression
145: Type conflict
146: Assignment of files not allowed
147: Label type incompatible with selecting expression
148: Subrange bounds must be scalar
149: Index type must be integer
150: Assignment to standard function is not allowed

151: Assignment to formal function is not allowed
152: No such field in this record
153: Type error in read
154: Actual parameter must be a variable
155: Control variable cannot be formal or nonlocal
156: Multidefined case label
157: Too many cases in case statement
158: No such variant in this record
159: Real or string tagfields not allowed
160: Previous declaration was not forward
161: Again forward declared
162: Parameter size must be constant
163: Missing variant in declaration
164: Substitution of standard proc/func not allowed
165: Multidefined label
166: Multideclared label
167: Undeclared label
168: Undefined label
169: Error in base set
170: Value parameter expected
171: Standard file was redeclared
172: Undeclared external file
174: Pascal function or procedure expected
175: Semaphore value parameter not allowed

182: Nested units not allowed
183: External declaration not allowed at this nesting level
184: External declaration not allowed in interface section
185: Segment declaration not allowed in unit
186: Labels not allowed in interface section
187: Attempt to open library unsuccessful
188: Unit not declared in previous uses declaration
189: 'Uses' not allowed at this nesting level
190: Unit not in library
191: No private files
192: 'Uses' must be in interface section
193: Not enough room for this operation
194: Comment must appear at top of program
195: Unit not importable
196: Must use LONGINT unit

201: Error in real number - digit expected
202: String constant must not exceed source line
203: Integer constant exceeds range
204: 8 or 9 in octal number
250: Too many scopes of nested identifiers
251: Too many nested proedures or functions
252: Too many forward references of procedure entries
253: Procedure too long
254: Too many long constants in this procedure
256: Too many external references
257: Too many externals
258: Too many local files
259: Expression too complicated

300: Division by zero
301: No case provided for this value
302: Index expression out of bounds
303: Value to be assigned is out of bounds
304: Element expression out of range
398: Implementation restriction
399: Implementation restriction

400: Illegal character in text
401: Unexpected end of input
402: Error in writing code file, not enough room
403: Error in reading include file
404: Error in writing list file, not enough room
405: Call not allowed in separate procedure
406: Include file not legal
407: I/O error in handling linker refs

B.4 UNIT NUMBERS

Version 3.Ø

NUMBER	VOLUME NAME
Ø	<empty>
1	CONSOLE
2	SYSTERM
4	Blocked Volume
5	Blocked Volume
6	PRINTER
7	RCONS1
8	REMOTE <serial port B>
9-14	Blocked Volumes
15	RCONS2
16	RTERM2
17	RCONS3
18	RTERM3
19	RCONS4
2Ø	RTERM4
21	RCONS5
22	RTERM5
23	RCONS6
24	RTERM6
25	RCONS7
26	RTERM7
27	PRINTR1
28..25	Winchester disk blocked volumes

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

APPENDIX C: P-MACHINE OPCODES AND OPERATOR EXECUTION TIMES

This appendix presents tables of P-machine opcodes and operator execution times. Table C-1 presents the opcodes, and Table C-2 presents the operator execution times. Table C-3 lists the P-codes in a Pascal-like metalanguage.

C-1. P-MACHINE OPCODES

Instructions are one byte long, followed by zero-to-three parameters. Most parameters specify one word of information and are one of five basic types.

UB Unsigned byte: high-order byte of parameter is implicitly zero.

SB Signed byte: high-order byte is sign extension of bit 7.

DB Don't care byte: can be treated as SB or UB, because the value is always in the range Ø..127.

B Big: this parameter is one byte long when used to represent values in the range Ø..127 and is two bytes when representing values in the range 128..32767. If the first byte is in the Ø..127 range, the high byte of the parameter is implicitly zero. Otherwise, bit 7 of the first byte is cleared and is used as the high order byte of the parameter. The second byte is used as the low-order byte.

W Word: the next two bytes (low byte first) are the parameter value.

These mnemonics are intended only for further understanding of P-code. Neither the Western Digital Corporation nor the University of California at San Diego provide P-code assembler software.

Table C-1. P-Machine Opcodes.

Mnemonic	Instruction Code	Parameters	Description
		Constant One Word Loads	
SLDC	Ø..31		Short Load Word Constant (Value Ø-31). Pushes the opcode, with high byte zero, onto the stack.
LDCN	152		Load Constant Nil (FCØØ). Pushes nil onto the stack.
LDCB	128	UB	Load Constant Byte. Pushes UB, with high byte zero, onto the stack.
LDCI	129	W	Load Constant Word. Pushes W onto the stack.
LCA	13Ø	B	Load Constant Address. Pushes the word address of the constant, with offset B in the constant word block, onto the stack.
		Local One Word Loads and Store	
SLDL1..16	32..47		Short Load Local Word. Fetches the word with offset 1..16 in the local activation record and pushes it on the stack.
LDL	135	B	Load Local Word. Fetches the word with offset B in the local activation record and pushes it on the stack.
LLA	132	B	Load Local Address. Fetches address of the word with offset B in the local activation record and pushes it on the stack.
STL	164	B	Store Local. Stores Tos into the word with offset B in the local activation record.

Table C-1. P-Machine Opcodes. (Continued)

Mnemonic	Instruction Code	Parameters	Description
			Global One Word Loads and Store
SLDO1..16	48..63		Short Load Global Word. Fetches the word with offset 1..16 in the base activation record and pushes it on the stack.
LDO	133	B	Load Global Word. Fetches the word with offset B in the base activation record and pushes it on the stack.
LAO	134	B	Load Global Address. Pushes the word address of the word with offset B in the base activation record.
SRO	165	B	Store Global Word. Stores Tos into the word with offset B in the base activation record.
			Intermediate One-Word Loads and Store
LOD	137	DB,B	Load Intermediate Word. DB indicates the number of static links to traverse to find the activation record to use. B is the offset within the activation record.
LDA	136	DB,B	Load Intermediate Address.
STR	166	DB,B	Store Intermediate Word.
			Indirect One-Word Loads and Store
STO	196		Store Indirect. Tos is stored into the word pointed to by Tos-1.

Table C-1. P-Machine Opcodes. (Continued)

Mnemonic	Instruction Code	Parameters	Description
			Extended One-Word Loads and Store
LDE	154	UB,B	Load Word Extended. Fetches the word with offset B in segment number UB and pushes it on the stack.
LAE	155	UB,B	Load Address Extended.
STE	217	UB,B	Store Word Extended.
			Multiple Word Loads and Stores (Sets and Reals)
LDC	131	B,UB	Load Multiple Word Constant. B is the offset within the constant word block, and UB is the number of words to load. Push the block onto the stack.
LDM	208	UB	Load Multiple Words. Tos is a pointer to the beginning of a block of UB words Push the block onto the stack.
STM	142	UB	Store Multiple Words. Tos is a block UB words, Tos-1 is a word pointer to a similar block. Transfer the block from the stack to the destination block.
			Byte Arrays
LDB	167		Load Byte. Push the byte (after zeroing high byte) pointed to by byte pointer Tos.
STB	200		Store Byte. Store byte Tos into the location specified by Byte Pointer Tos-1.

Table C-1. P-Machine Opcodes. (Continued)

Mnemonic	Instruction Code	Parameters	Description
			Record and Array Indexing and Assignment
MOV	197	B	Move Words. Tos is a source pointer to a block of B words, Tos-1 is a destination pointer to a similar block. Transfer the block from the source to the destination.
SINDØ..7	12Ø..127		Short Index and Load Word. Indexes the word pointer Tos by Ø..7 words, and pushes the word pointed to by the result.
IND	23Ø	B	Static Index and Load Word. Index the word pointer Tos by B words, and push the word pointed to.
INC	231	B	Increment Field Pointer. The word pointer Tos is indexed by B words and the resultant pointer is pushed.
IXA	215	B	Index Array. Tos is an integer index, Tos-1 is the array base word pointer, and B is the size (in words) of an array element. A word pointer to the indexed element is pushed.
IXP	216	UB1,UB2	Index Packed Array. Tos is an integer index, Tos-1 is the array base word pointer. UB1 is the number of elements-per-word, and UB2 is the field-width (in bits). Compute and push a packed field pointer.
LDP	2Ø1		Load A Packed Field. Push the field described by the packed field pointer, Tos.
STP	2Ø2		Store Into A Packed Field. Tos is the data, Tos-1 is a packed field pointer. Store Tos into the field described by Tos-1.

Table C-1. P-Machine Opcodes. (Continued)

Mnemonic	Instruction Code	Parameters	Description
			Logicals
LAND	161		Logical AND. AND Tos into Tos-1.
LOR	160		Logical OR. OR Tos into Tos-1.
LNOT	229		Logical NOT.Take one's complement of Tos.
BNOT	159		Boolean NOT.
LEUSW	180		Compare Unsigned Word <=. Compare unsigned word of Tos-1 to unsigned word of Tos and push true or false.
GEUSW	181		Compare Unsigned Word >=. Compare unsigned word of Tos-1 to unsigned word of Tos and push true or false.
			Integers
ABI	224		Absolute Value of Integer. Take absolute value of integer Tos.
NGI	225		Negate Integer. Take the two's complement of Tos.
DUP1	226		Copy Integer. Duplicate one word on Tos.
ADI	162		Add Integers. Add Tos and Tos-1.
SBI	163		Subtract Integers. Subtract Tos from Tos-1.
MPI	140		Multiply Integers. Multiply Tos and Tos-1.
DVI	141		Divide Integers. Divide Tos-1 by Tos and push quotient.
MODI	143		Modulo Integers. Divide Tos-1 by Tos and push the remainder.

Table C-1. P-Machine Opcodes. (Continued)

Mnemonic	Instruction Code	Parameters	Description
		Integers (Continued)	
CHK	203		Check Against Subrange Bounds. Insure that Tos-1 < = Tos-2 < = Tos, leaving Tos-2 on the stack. If conditions are not satisfied a run-time error occurs.
EQUI	176		Compare Integer =. Compare Tos-1 to Tos and push true or false.
NEQI	177		Compare Integer <>. Compare Tos-1 to to Tos and push true or false.
LEQI	178		Compare Integer <=. Compare Tos-1 to to Tos and push true or false.
GEQI	179		Compare Integer >=. Compare Tos-1 to to Tos and push true or false.
	Reals (All Over/Underflows Cause a Run-Time Error)		
FLT	204		Float Top-of-Stack. The integer Tos is converted to a floating point number.
TNC	190		Truncate Real. The real Tos is truncated and converted to an integer. A run-time error results if the real is outside of permissible integer values.
RND	191		Round Real. The real Tos is rounded and converted to an integer. A run-time error results if the real is outside of permissible integer values.

Table C-1. P-Machine Opcodes. (Continued)

Mnemonic	Instruction Code	Parameters	Description
			Reals (All Over/Underflows Cause a Run-time Error) (Continued)
ABR	227		Absolute Value of Real. Take the absolute value of the real Tos.
NGR	228		Negate Real. Negate the Real Tos.
DUP2	198		Copy Real. Duplicate two words on Tos.
ADR	192		Add Reals. Add Tos and Tos-1.
SBR	193		Subtract Reals. Subtract Tos from Tos-1.
MPR	194		Multiply Reals. Multiply Tos and Tos-1.
DVR	195		Divide Reals. Divide Tos-1 by Tos.
EQUREAL	205		Compare Real =. Compare Tos-1 to Tos and push true or false.
LEQREAL	206		Compare Real <=. Compare Tos-1 to Tos and push true or false.
GEQREAL	207		Compare Real >=. Compare Tos-1 to Tos and push true or false.
			Sets
ADJ	199	UB	Adjust Set. The set Tos is forced to occupy UB words, either by expansion (putting zeroes "between" Tos and Tos-1) or compression (chopping of high words of set), and its length word is discarded.

Table C-1. P-Machine Opcodes. (Continued)

Mnemonic	Instruction Code	Parameters	Description
SRS	188		Build Subrange Set. The integers Tos and Tos-1 are checked to insure that 0<=Tos<=4079 and 0<=Tos-1<=4079, otherwise a run-time error occurs. The set [Tos-1..Tos] is pushed. (The set [ ] is pushed if Tos-1 > Tos.)
INN	218		Set Membership. See if integer Tos-1 is in set Tos, pushing true or false.
UNI	219		Set Union. The union of sets Tos and Tos-1 is pushed. (Tos or Tos-1.)
INT	220		Set Intersection. The intersection of sets Tos and Tos-1 is pushed. (Tos and Tos-1.)
DIF	221		Set Difference. The difference of sets Tos-1 and Tos is pushed. (Tos-1 and not Tos.)
EQUPWR	182		Set Compare =.
LEQPWR	183		Set Compare <= (Subset of).
GEQPWR	184		Set Compare >= (Superset of).
		Byte Arrays	
EQUBYT	185	B	Byte Array Compare =.
LEQBYT	186	B	Byte Array Compare <=.
GEQBYT	187	B	Byte Array Compare >=.

Table C-1. P-Machine Opcodes. (Continued)

Mnemonic	Instruction Code	Parameters	Description
		Jumps	
UJP	138	SB	Unconditional Jump. Jump to location with offset SB from the current location.
FJP	212	SB	False Jump. Jump to location with offset SB from the current location if Tos is false.
EFJ	210	SB	Equal False Jump. Jump to location with offset SB from the current location if integer Tos <> Tos-1.
NFJ	211	SB	Not Equal False Jump. Jump to location with offset SB from the current location if integer Tos = Tos-1.
UJPL	139	W	Unconditional Long Jump. Jump unconditionally to location with offset W from the current location.
FJPL	213	W	False Long Jump. Jump to location with offset W from the current location if Tos is false.
XJP	214	B	Case Jump. The word with offset B in the constant word block is W1, the minimum index of the Case Table. The word with offset B+1 in the constant word block is W2, the maximum index of the Case Table. The Case Table starts in location with offset B+2 in the constant word block and has a length of W2-W1+1 words. Tos is an index. If tos is in the range of W1..W2, then fetch the contents of the location with tos index in the Case Table and jump to the location with this offset from the current location.

Table C-1. P-Machine Opcodes. (Continued)

Mnemonic	Instruction Code	Parameters	Description
			Procedure and Function Calls and Returns
CPL	144	UB	Call Local Procedure. Call procedure UB, which is an immediate child of the currently executing procedure and in the same segment. Static link of MSCW is set to old MP.
CPG	145	UB	Call Global Procedure. Call procedure UB, which is at the outer most lex level and in the same segment. The static link of the MSCW is set to base.
CPI	146	DB,UB	Call Intermediate Procedure. Call procedure UB, which is at lex level DB less than the currently executing procedure and in the same segment. Use that activation record's static link as the static link of the new MSCW.
CXL	147	UB1,UB2	Call Local External Procedure. Call procedure UB2 which is an immediate child of the currently executing procedure and in the segment UB1.
CXG	148	UB1,UB2	Call Global External Procedure. Call procedure UB2 which is at the outer most lex level and in the segment UB1.
CXI	149	UB1,DB,UB2	Call Intermediate External Procedure. Call procedure UB2 which is at lex level DB less than the currently executing procedure, and in the segment UB1.
CPF	151		Call Formal Procedure. Tos contains segment number and procedure number and Tos-1 contains static link for the called procedure.

Table C-1. P-Machine Opcodes. (Continued)

Mnemonic	Instruction Code	Parameters	Description
			Procedure and Function Calls and Returns (Continued)
RPU	150	B	Return From User Procedure. Static link is discarded, MP is reset from MSDYN, IPC is also reset from MSIPC. If segment number is not zero, segment pointer is set from segment dictionary. Stack pointer is decremented by B.
LSL	153	DB	Load Static Link Onto Stack. DB indicates the number of static links to traverse to get the static link to load.
			System Control
SIGNAL	222		Signal. Tos is a semaphore address. Signal this semaphore.
WAIT	223		Wait on Semaphore. Tos is a semaphore address. Wait on this semaphore.
LPR	157		Load Processor Register. Tos is a register #. (If it is positive it is one of the TIB registers. If not -1 is the current task pointer, -2 is the segment dictionary pointer and -3 is the ready queue pointer.) Load contents of this register on top of stack.
SPR	209		Store Processor Register. Tos-1 is a register number (same definition as LPR). Store Tos in this register.

Table C-1. P-Machine Opcodes. (Continued)

Mnemonic	Instruction Code	Parameters	Description
			Debugger
BPT	158		Break Point.
RBP	159	B	Return From Breakpoint. This acts like an RPU operator. The stack pointer is decremented by B.
			Miscellaneous
NOP	156		No Operation.
SWAP	189		Swap word Top-of-Stack with word Top-of-Stack-1.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

C-2. PASCAL MICROENGINE OPERATOR EXECUTION TIMES

Table C-2 presents the execution time of all 3.Ø P-code operators. Any P-code operator is made up of several operations. Any one of these operations would normally be considered as one machine-language operator on a non-stack machine. Therefore, P-code operator timings are not comparable to nonstack-machine-operator timings.

The operators are grouped by operation. The left-hand column contains the P-code mnemonic, followed by the 8-bit opcode for that P-code. Next, the P-code parameters zero to three are given. All execution times are in microseconds and were measured on an ME16ØØ running at 2.5 mhz.

Many of the P-code execution times are data dependent. For this table, the best and worst times are listed with comments describing how the values relate to the operands of the instruction. For some P-codes, the execution time between the best and worst is equally probable, depending on the execution environment. However, for some of these data-dependent P-codes, the execution times near the best case values are more probable than those of the worst case. For example, all operators that require static link traversal (LOD, LDA, STR, CPI, CXI, and LSL) traverse one-to-four links. In fact, compiler enforced restrictions disallow traversals of more than eight links. Thus, the worst case execution time for any of those P-codes, while theoretically possible, can never occur.

Under the mnemonic for each P-code is a notational description of the P-machine stack both before and after the execution of the P-code. A stack status description consists of a single pair of enclosing brackets ([]). The stack status on the left side of the colon represents the status prior to execution of the P-code, while the stack status on right of the colon represents the status following the execution of the P-code. Within the brackets, the stack grows from left to right, with individual operands seperated by commas. Operands within stack status descriptions are of the following types:

activation	-	a block of four, 16-bit words representing the record of activation of a procedure or function (MSCW).
addr	-	a 16-bit word address.
bool	-	a 16-bit value representing a Pascal BOOLEAN. The low-order bit signifies the Boolean value, all other bits are Ø. A value of Ø represents FALSE; a value of 1 represents TRUE.
byte-ptr	-	two, 16-bit values representing the address of an 8-bit byte.

func-result	-	either 1 or 2 16-bit values representing the result of a function left on the stack when returning from a function. No words are left on return from a procedure.
int	-	a 16-bit two's complement Pascal INTEGER.
nil	-	a 16-bit value representing a NIL pascal pointer.
pack-ptr	-	a "packed field pointer": three, 16-bit values defining the address of field of a packed variable. The values, from highest to lowest stack position, are 1) the rightmost bit # of the packed field, 2) the field width in bits and 3) the address of the word containing the field.
param	-	a block of 16-bit words representing the values of the parameters being passed to a procedure or function.
real	-	two, 16-bit values representing a Pascal REAL. One value contains the sign, exponent and high-order mantissa bits, the other value contains the low-order mantissa bits.
seg#/proc#	-	a 16-bit word containing 2, 8-bit bytes. The high byte is the segment number; the low byte, the procedure number of a procedure or function being invoked via P-code CPF.
set	-	a block of 1..256, 16-bit words representing a Pascal SET. The highest word in the set defines the number of words in the block of words below.
word	-	a 16-bit value.
word-block	-	a block of 2..255, 16-bit words.

All P-code parameters are one of five basic types :

- UB - "Unsigned byte" : value in the range Ø..255, high-order byte is implicitly zero.
- SB - "Signed byte" : value in the range -128..127, high-order byte is implicitly sign extension of bit 7.
- DB - "Don't care byte" : value in the range Ø..127, high-order byte is implicitly Ø.
- B - "Big" : one byte long when used to represent values in the range Ø..127; two bytes long when used to represent values in the range 128..32767. If the first byte is in the range Ø..127, the high byte is implicitly Ø. Otherwise, bit 7 of the first byte is cleared, and the first byte is used as the high-order byte of the parameter. The second byte is used as the low-order byte.
- W - "Word" : two-byte value in the range Ø..32767. The first byte is used as the low byte of the parameter.

Table C-2. Operator Execution Times.

Mnemonic	Opcode	Parameters	Time	Remarks
			Constant One-Word Loads	
SLDCØ..31 [] : [word]	Ø..31		2.8	
LDCN [] : [nil]	152		6.4	
LDCB [] : [word]	128	UB	5.6	
LDCI [] : [word]	129	W	8.4	
LCA [] : [addr]	13Ø	B	8.Ø	
		Local One-Word Loads and Stores		
SLDL1..16 [] : [word]	32..47		6.4	
LDL [] : [word]	135	B	9.6	
LLA [] : [addr]	132	B	7.6	
STL [word] : []	164	B	9.6	

Table C-2. Operator Execution Times. (Continued)

Mnemonic	Opcode	Parameters	Time	Remarks
		Global One-Word Loads and Stores		
SLDO1..16 [] : [word]	48..63		7.2	
LDO [] : [word]	133	B	10.0	
LAO [] : [addr]	134	B	8.0	
SRO [word] : []	165	B	13.2	
		Intermediate One-Word Loads and Stores		
LOD [] : [word]	137	DB, B	17.2 to 423.6	17.2 + 3.2(DB).
LDA [] : [addr]	136	DB, B	15.2 to 421.6	15.2 + 3.2(DB).
STR [word] : []	166	DB, B	16.8 to 423.2	16.8 + 3.2(DB).
		Indirect One-Word Loads and Stores		
STO [addr,word] : []	196		8.0	

Table C-2. Operator Execution Times. (Continued)

Mnemonic	Opcode	Parameters	Time	Remarks
		Extended One-Word Loads and Stores		
LDE [] : [word]	154	UB, B	26.8	
LAE [] : [addr]	155	UB, B	24.8	
STE [word] : []	217	UB, B	26.Ø	
		Multiple-Word Loads and Stores (Sets and Reals)		
LDC [] : [word-block]	131	B, UB	18.Ø to 1Ø38.Ø	18.Ø + 4.Ø(UB)
LDM [addr] : [word-block]	2Ø8	UB	1Ø.4 to 154Ø.4	1Ø.4 + 6.Ø(UB)
STM [word-block,addr] : []	142	UB	12.4 to 1532.4	12.4 + 6.Ø(UB)
			Byte Arrays	
LDB [byte-ptr] : [word]	167		12.Ø	
STB [byte-ptr,word] : []	2ØØ		13.6	

Table C-2. Operator Execution Times. (Continued)

Mnemonic	Opcode	Parameters	Time	Remarks
		Record and Array Indexing and Assignment		
MOV [addr,addr] : []	197	B	13.2 to 196615.2	13.2 + 6.0(B).
SIND0..7 [addr] : [word]	120..127		8.4	
IND [addr] : [word]	230	B	12.4	
INC [addr] : [addr]	231	B	9.6	
IXA [addr,word] : [addr]	215	B	9.6 to 56.8	9.6 is best case time (index (TOS) is 0), time increases when index exceeds B. Worst case time (56.8) arrives with array element size (B) of 16384.
IXP [addr,word] : [pack-ptr]	216	UB1, UB2		Times indicated are for indices (TOS) in the 1st word of the array. Values in parenthesis indicate index range for which the corresponding time is obtained. ALL times are 73.6 larger if index is not in 1st word.
LDP [pack-ptr] : [word]	201		18.4 to 50.4	18.4 + 2.0(fieldwidth) + 2.0(right bit #).
STP [pack-ptr,word] : []	202		20.4 to 64.4	20.4 + 2.0(fieldwidth) + 2.8(right bit #).

IXP:

Elements per word	Best time	Worst time
3	27.6(0)	37.2(2)
4	27.6(0)	38.8(3)
5	27.6(0)	39.6(4)
8	27.6(0)	38.0(3..7)
16	27.6(0)	35.6(2..15)

Table C-2. Operator Execution Times. (Continued)

Mnemonic	Opcode	Parameters	Time	Remarks
			Logicals	
LAND	161	[word,word] : [word]	8.0	
LOR	160	[word,word] : [word]	8.0	
LNOT	229	[word] : [word]	5.2	
BNOT	159	[bool] : [bool]	6.0	
LEUSW	180	[word,word] : [bool]	9.6 | 10.4	TRUE | FALSE
GEUSW	181	[word,word] : [bool]	9.6 | 10.4	TRUE | FALSE

Table C-2. Operator Execution Times. (Continued)

Mnemonic	Opcode	Parameters	Time	Remarks
			Integers	
ABI	224	[int] : [int]	4.8 | 6.Ø	pos parm | neg parm
NGI	225	[int] : [int]	5.2	
DUP1	226	[word] : [word,word]	5.2	
ADI	162	[int,int] : [int]	8.Ø	
SBI	163	[int,int] : [int]	8.Ø	
MPI	14Ø	[int,int] : [int]	5.2 to 35.2	Best case (5.2) is n * Ø, worst case is -i * -j where i, j are large values. Typical time will be around 28.Ø.
DVI	141	[int,int] : [int]	8.4 | 89.2 | 91.2	Ø div i | positive result | negative result.
MODI	143	[int,int] : [int]	9.2 | 89.2 | 92.8	Ø mod i | positive result | negative result.
CHK	2Ø3	[int,int,int] : [int]	14.4	
EQUI	176	[int,int] : [bool]	9.6 | 1Ø.4	TRUE | FALSE
NEQI	177	[int,int] : [bool]	9.6 | 1Ø.4	TRUE | FALSE
LEQI	178	[int,int] : [bool]	1Ø.4 | 11.2	TRUE | FALSE
GEQI	179	[int,int] : [bool]	1Ø.4 | 11.2	TRUE | FALSE

Table C-2. Operator Execution Times. (Continued)

<table>
<tr><th>Mnemonic</th><th>Opcode</th><th>Parameters</th><th>Time</th><th>Remarks</th></tr>
<tr><td colspan="5">Reals</td></tr>
<tr><td>FLT</td><td>2Ø4</td><td>[int] : [real]</td><td>1Ø.8 | 14.8 to 46.8</td><td>TOS = Ø |
TOS <> Ø :
44.8 -
2.Ø(trunc(lg(abs(TOS)))) + C.
C = 2.Ø if TOS < Ø
C = Ø.Ø otherwise.</td></tr>
<tr><td>TNC</td><td>19Ø</td><td>[real] : [int]</td><td>12.4 |
15.6 |
5Ø.Ø to 5Ø.8 |

24.Ø to 48.8</td><td>TOS = Ø.Ø |
Ø.Ø < abs(TOS) < Ø.5 |
Ø.5 <= abs(TOS) < 1.Ø :
5Ø.Ø + C |
abs(TOS) >= 1.Ø :
48.Ø -
Ø.8(trunc(lg(abs(TOS)))) + C.
C = Ø.8 if TOS < Ø.Ø,
C = Ø.Ø otherwise.</td></tr>
<tr><td>RND</td><td>191</td><td>[real] : [int]</td><td>12.4 |
15.6 |
52.4 to 53.2 |

24.8 to 49.6</td><td>TOS = Ø.Ø |
Ø.Ø < abs(TOS) < Ø.5 |
Ø.5 <= abs(tos) < 1.Ø :
52.4 + C |
48.8 -
Ø.8(trunc(lg(abs(TOS)))) + C.
C = Ø.8 if TOS < Ø.Ø,
C = Ø.Ø otherwise.</td></tr>
<tr><td>ABR</td><td>227</td><td>[real] : [real]</td><td>5.2</td><td></td></tr>
<tr><td>NGR</td><td>228</td><td>[real] : [real]</td><td>5.2</td><td></td></tr>
<tr><td>DUP2</td><td>198</td><td>[word,word] : [word,word,word,word]</td><td>12.Ø</td><td></td></tr>
<tr><td>ADR</td><td>192</td><td>[real,real] : [real]</td><td>18.8 |
6Ø.8 to 152.8</td><td>TOS-1 = Ø.Ø |
Range of times represents difference in exponents of TOS and TOS-1. As the difference increases, the time increases until the difference exceeds the width of the mantissa.</td></tr>
</table>

Table C-2. Operator Execution Times. (Continued)

Mnemonic	Opcode	Parameters	Time	Remarks
		Reals Continued		
SBR [real,real] : [real]	193		19.2 | 64.4 to 152.Ø	TOS-1 = Ø.Ø | Times vary for same reasons as ADR.
MPR [real,real] : [real]	194		26.4 | 159.4 to 177.8	TOS-1 = Ø.Ø | Time is a function of the operands.
DVR [real,real] : [real]	195		32.4 | 14Ø.6 to 293.8	TOS = Ø.Ø | Time is a function of the operands.
EQUREAL [real,real] : [bool]	2Ø5		16.4 | 14.8 | 18.4	TRUE result | FALSE in 1st word | FALSE in 2nd word.
LEQREAL [real,real] : [bool]	2Ø6		16.4 | 16.Ø to 2Ø.4 | 16.8 to 22.Ø	TRUE (TOS = TOS-1) | TRUE (TOS < TOS-1) : 16.Ø + B + C, B = Ø.8 if "pos < pos", B = Ø.Ø otherwise, C = 3.6 if equal in 1st word, Ø.Ø otherwise | FALSE (TOS > TOS-1) : 16.8 + B + C, B = 1.6 if "pos < pos", B = Ø.Ø otherwise, C = 3.6 if equal in 1st word, Ø.Ø otherwise.
GEQREAL [real,real] : [bool]	2Ø7		16.4 | 16.Ø to 2Ø.4 | 16.Ø to 2Ø.4	TRUE (TOS = TOS-1) | TRUE (TOS > TOS-1) : 16.Ø + B + C, B = Ø.8 if "pos > pos", B = Ø.Ø otherwise, C = 3.6 if equal in 1st word, Ø.Ø otherwise. FALSE (TOS < TOS-1) : 16.Ø + B + C, B = Ø.8 if "pos > pos", B = Ø.Ø otherwise, C = 3.6 if equal in 1st word, Ø.Ø otherwise.

Table C-2. Operator Execution Times. (Continued)

<table>
<tr><th>Mnemonic</th><th>Opcode</th><th>Parameters</th><th>Time</th><th>Remarks</th></tr>
<tr><td></td><td></td><td></td><td>Sets</td><td></td></tr>
<tr><td>ADJ
[set] : [word-block]</td><td>199</td><td>UB</td><td>14.4 |
13.6 to 1747.6 |</td><td>words(TOS) = UB |
set expansion :
13.6 + 6.8(UB) |</td></tr>
<tr><td></td><td></td><td></td><td>16.4 to 1431.6</td><td>set compression :
16.4 +
5.6(words(TOS)) +
2.8(UB - words(TOS))</td></tr>
<tr><td>SRS
[int,int] : [set]</td><td>188</td><td></td><td>18.0 |
50.4 to 110.4 |</td><td>null set (TOS-1 < TOS) |
1 word set :
50.4 +
2.0(TOS-1) +
2.0(TOS) |</td></tr>
<tr><td></td><td></td><td></td><td>52.4 to 114.0 |</td><td>2 word set :
52.4 +
2.0(TOS mod 16) +
2.0(TOS-1 mod 16) + C,
C = 1.6 if TOS > 15,
C = 0.0 otherwise |</td></tr>
<tr><td></td><td></td><td></td><td>56.4 to 1023.6</td><td>all others :
45.6 +
3.6((TOS div 16) + 1) +
2.0(TOS mod 16) +
2.0(TOS-1 mod 16) - B,
B = 0.4 if
((TOS div 16) -
(TOS-1 div 16)) < 2,
B = 0.0 otherwise.</td></tr>
<tr><td>INN
[int,set] : [bool]</td><td>218</td><td></td><td>18.4 |</td><td>TOS-1 outside bounds of
set TOS |</td></tr>
<tr><td></td><td></td><td></td><td>22.8 to 52.8</td><td>22.8 + 2.0(TOS-1 mod 16)</td></tr>
<tr><td>UNI
[set,set] : [set]</td><td>219</td><td></td><td>6.6 |
29.2 to 1756.4 |</td><td>TOS is null set |
TOS-1 is null set :
22.4 +
6.8(words(TOS)) |</td></tr>
<tr><td></td><td></td><td></td><td>19.6 to 1848.4 |</td><td>words(TOS) <= words(TOS-1):
12.4 +
7.2(words(TOS)) |</td></tr>
<tr><td></td><td></td><td></td><td>58.8 to 3475.2</td><td>words(TOS) > words(TOS-1) :
24.0 +
14.0(words(TOS)) +
6.8(words(TOS) -
words(TOS-1))</td></tr>
</table>

Table C-2. Operator Execution Times. (Continued)

Mnemonic	Opcode	Parameters	Time	Remarks
		Sets Continued		
INT [set,set] : [set]	220		11.6	both sets null
			12.0	only TOS null
			22.4 to 1851.2	words(TOS) >= words(TOS-1) : 15.2 + 7.2(words(TOS))
			26.6 to 1848.2	words(TOS) < words(TOS-1) : 16.6 + 7.2(words(TOS) + 2.8(words(TOS-1) - words(TOS))
DIF [set,set] : [set]	221		6.0	TOS is null set
			12.0	TOS-1 is null set
			21.2 to 1850.0	words(TOS) <= words(TOS-1) : 14.0 + 7.2(words(TOS))
			20.8 to 1842.4	words(TOS) > words(TOS-1) : 13.6 + 7.2(words(TOS-1))
EQUPWR [set,set] : [bool]	182		23.6 to 1954.0	16.0 + 7.6(N) + 4.0(D) + C + B. N = # words compared to assert FALSE. 0 < N < words in smaller set D = # words examined in larger set (beyond size of smaller set) to assert FALSE. 0 <= D <= (size of larger set) - N C = 2.0 if D <> 0 and result is TRUE, 0.0 otherwise. B = 0.0 if words(TOS) >= words(TOS-1), else 1.2 if result TRUE else 0.8 if result FALSE

Table C-2. Operator Execution Times. (Continued)

Mnemonic	Opcode	Parameters	Time	Remarks
			Sets Continued	
LEQPWR [set,set] : [bool]	183		24.4 to 2158.Ø	words(TOS) >= words(TOS-1) : 16.Ø + 8.4(N)
			3Ø.Ø to 2175.2	words(TOS-1) > words(TOS) : 17.2 + 8.4(N) + 4.Ø(D) + C N = same as EQUPWR D = same as EQUPWR C = Ø.4 if D <> Ø and result is TRUE, Ø.Ø otherwise
GEQPWR [set,set] : [bool]	184		31.2 to 218Ø.8	words(TOS-1) >= words(TOS) : 21.6 + 8.4(N) + C + B C = 1.2 if result is TRUE, else Ø.Ø B = Ø.Ø if sets same size, else Ø.4 if result TRUE, else 1.2 if result FALSE
			29.2 to 2176.4	words(TOS) > words(TOS-1) : 2Ø.8 + 8.4(N) + 4.Ø(D) + C N = same as EQUPWR D = same as EQUPWR C = 2.Ø if D <> Ø and result is TRUE, Ø.Ø otherwise

Table C-2. Operator Execution Times. (Continued)

Mnemonic	Opcode	Parameters	Time	Remarks
			Byte Arrays	
EQUBYT [addr,addr] : [bool]	185	B	29.6 to 170404.8	TRUE result : 19.2 + 10.4((B+1) div 2) + 2.8((B+1) mod 2)
			21.8 to 170397.0	FALSE result : 11.4 + 10.4((D+1) div 2) + 2.8((D+1) mod 2) D = # bytes compared to assert FALSE.
LEQBYT [addr,addr] : [bool]	186	B	28.8 to 170404.0	EQUAL (TRUE) result : 18.4 + 10.4((B+1) div 2) + 2.8((B+1) mod 2)
			27.2 to 170402.4	LESS (TRUE) result : 16.8 + 10.4((L+1) div 2) + 2.8((L+1) mod 2) L = # bytes compared to assert LESS
			28.0 to 170403.2	GREATER (FALSE) result : 17.6 + 10.4((G+1) div 2) + 2.8((G+1) mod 2) G = # bytes compared to assert GREATER.
GEQBYT [addr,addr] : [bool]	187	B	28.8 to 170404.0	EQUAL (TRUE) result : 18.4 + 10.4((B+1) div 2) + 2.8((B+1) mod 2)
			31.6 to 170406.8	GREATER (TRUE) result : 21.2 + 10.4((G+1) div 2) + 2.8((G+1) mod 2) G = # bytes compared to assert GREATER
			32.4 to 170407.6	LESS (FALSE) result : 22.0 + 10.4((L+1) div 2) + 2.8((L+1) mod 2) L = # bytes compared to assert LESS.

Table C-2. Operator Execution Times. (Continued)

Mnemonic	Opcode	Parameters	Time	Remarks
			Jumps	
UJP [] : []	138	SB	12.4	
FJP [bool] : []	212	SB	16.8 | 10.8	jump | no jump
EFJ [int,int] : []	210	SB	19.2 | 11.8	jump | no jump
NFJ [int,int] : []	211	SB	19.2 | 12.0	jump | no jump
UJPL [] : []	139	W	12.8	
FJPL [bool] : []	213	W	18.8 | 10.0	jump | no jump
XJP [int] : []	214	B	32.0 | 29.2 | 34.0	jump | TOS < min index | TOS > max index

Table C-2. Operator Execution Times. (Continued)

Mnemonic	Opcode	Parameters	Time	Remarks
		Procedure and Function Calls and Returns		
CPL	144	UB	45.6	
[param] : [activation]				
CPG	145	UB	44.8	
[param] : [activation]				
CPI	146	DB, UB	56.8 to 450.0	53.6 + 3.2(DB)
[param] : [activation]				
CXL	147	UB1, UB2	64.4	
[param] : [activation]				
CXG	148	UB1, UB2	63.2	
[param] : [activation]				
CXI	149	UB1, DB, UB2	76.4 to 469.6	73.2 + 3.2(DB)
[param] : [activation]				
CPF	151		75.6	
[param,addr,seg#/proc#] : [activation]				
RPU	150	B	26.0	
[activation] : [func-result]				
LSL	153	DB	15.6 to 408.8	12.4 + 3.2(DB)
[] : [addr]				

Table C-2. Operator Execution Times. (Continued)

<table>
<tr><th>Mnemonic</th><th>Opcode</th><th>Parameters</th><th>Time</th><th>Remarks</th></tr>
<tr><td colspan="5">System Control</td></tr>
<tr><td>SIGNAL</td><td>222</td><td>[addr] : []</td><td>14.8 |
18.Ø |
52.Ø |

134.8</td><td>waitq nil, count > Ø |
waitq nil, count = Ø |
waitq non-nil, no taskswitch |
waitq non-nil, taskswitch performed</td></tr>
<tr><td>WAIT</td><td>223</td><td>[addr] : []</td><td>11.6 |
9Ø.8</td><td>count > Ø, no wait |
count = Ø, 9Ø.8 is time to taskswitch to another task.</td></tr>
<tr><td>LPR</td><td>157</td><td>[int] : [word]</td><td>8.4 |
55.2</td><td>TOS < Ø |
TOS >= Ø</td></tr>
<tr><td>SPR</td><td>2Ø9</td><td>[int,word] : []</td><td>8.4 |
53.2 |
54.8</td><td>TOS - 1 = -2, -3 |
TOS - 1 = -1 |
TOS - 1 >= Ø</td></tr>
<tr><td colspan="5">Debugger</td></tr>
<tr><td>BPT</td><td>158</td><td>[] : [activation]</td><td>- - -</td><td>time for this operator is comparable to the time for CXG. BPT unconditionally calls execution error procedure, resulting in a halt of execution.</td></tr>
<tr><td colspan="5">Miscellaneous</td></tr>
<tr><td>NOP</td><td>156</td><td>[] : []</td><td>3.6</td><td></td></tr>
<tr><td>SWAP</td><td>189</td><td>[word,word] : [word,word]</td><td>12.4</td><td></td></tr>
</table>

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

C-3. P-MACHINE DESCRIPTION METALANGUAGE

This appendix presents the III.Ø P-code operators in a Pascal-like notation. Pointer expressions are allowed. For example sp^.i is the contents of the memory location the top of stack register is pointing at taken as an integer. The expression (sp+1)^.i is one memory cell above the the sp register taken as an integer. The notation i<x:y> means take the field starting from bit position x for y bits. Table C-3 shows the P-code operators in a Pascal-like metalanguage.

The record declarations used are close to those used by the Western Digital MicroEngine operating system. The declarations follow.

```
const
    version     = 'BØ';   { Version of this document }
    mscw_sz     = 4;      { Size of mark stack control word in words}
    real_sz     = 2;      { Size of reals in words}
    bset_sz     = 4Ø8Ø;   { Max size of sets in bits}
    iset_sz     = 255;    { Max size of sets in words}
    word_sz     = 16;     { Size of word in bits}
    NIL         = -1Ø24;  { Representation for nil pointer}
type
   object_type     = (int_obj, real_obj, byte_obj, bool_obj, set_obj,
                       ptr_obj, sv_obj, sem_obj, mscw_obj, tib_obj);
   byte            = Ø..255;
   sibp            = ^sib;
   sibvec          = array [Ø..127] of sibp;
   sib             = record { segment info block }
                        segbase: memp;     { memory address of seg }
                        segleng: integer;  { # words in segment }
                        segrefs: integer;  { active calls }
                        segaddr: integer;  { absolute disk address }
                        segunit: integer;  { physical disk unit }
                        prevsp : memp;     { SP saved by getseg for relseg }
                      end { sib } ;
   mscwp           = ^mscw;
   mscw            = packed record  { mark stack control word }
                        msstat: mscwp;    { lexical parent pointer }
                        msdynl: mscwp;    { ptr to caller's mscw }
                        msipc:  integer;  { byte index in return code seg }
                        msseg:  byte;     { seg # of caller code }
                        msflag: byte
                      end { mscw } ;
```

```
  tibp            = ^tib;
  tib             = packed record { Task Information Block }
                      waitq: tibp;        { QUEUE LINK FOR SEMAPHORES }
                      prior: byte;        { TASK'S CPU PRIORITY }
                      flags: byte;        { STATE FLAGS...reserved }
                      splow: memp;        { LOWER STACK POINTER LIMIT }
                      spupr: memp;        { UPPER LIMIT ON STACK }
                      sp: memp;          { ACTUAL TOP-OF-STACK POINTER }
                      mp: mscwp;          { ACTIVE PROCEDURE MSCW PTR }
                      bp: mscwp;          { BASE ADDRESSING ENVIRONMENT PTR }
                      ipc: integer;       { BYTE PTR IN CURRENT CODE SEG }
                      segb: memp;         { PTR TO SEG CURRENTLY RUNNING }
                      hangp: semp;        { WHICH TASK IS WAITING ON }
                      iorslt : integer;   { Result of last I/O call. }
                      sibs: ^sibvec       { ARRAY OF SIBS FOR 128..255 }
                    end { TIB } ;
  memp            = ^memtrix;
  register        = memp;
  memtrix         = record
                      case object_type of
                        int_obj  : (i : integer);
                        bool_obj : (bool: boolean);
                        real_obj : (r : real);  { Standard, IEEE format }
                        byte_obj : (b : packed array [Ø..maxint-1] of byte);
                        set_obj  : (sz : integer;
                                    sb :packed array [Ø..bset_sz-1] of boolean);
                        ptr_obj  : (p : memp);
                        mscw_obj : (m : mscw);
                        tib_obj  : (t : tib);
                        sv_obj   : (sv: sibvec);
                        sem_obj  : (count: integer;
                                    waitq: tibp);
                    end {record memtrix};

var
  pc,       { program counter }
  sp,       { points at top item on stack which grows toward low memory }
  mp,       { points at current mark stack control word }
  bp,       { points at global mark stack control word }
  segb,     { points at base of currently executing code segment }
  ctp,      { points at TIB for currently executing task }
  rq,       { points at list of TIB's for ready to run tasks }
  ssv:      { points at system segment vector }
      register;

  lm,
  lsv,
  src,
  dst:        memp; { General use temporaries }
```

Table C-3. P-Code Operators in a Pascal-like Metalanguage.

Mnemonic	Op-Code in Hex		Semantics
			Constant One Word Loads
SLDCi	ØØ..1F		Short Load Word Constant. sp := sp - 1; sp^.i := i{Ø..31}
LDCN	98		Load Constant Nil. sp := sp - 1; sp^.p := NIL
LDCB	8Ø	UB	Load Constant Byte. sp := sp - 1; sp^.i := UB
LDCI	81	W	Load Constant Word. sp := sp - 1; sp^.i := W
LCA	82	B	Load Constant Address sp := sp - 1; sp^.p := segb + B
			Local One Word Loads and Store
SLDLi	2Ø..2F		Short Load Local Word. sp := sp - 1; sp^.i := (mp + mscw_sz - 1 + i{1..16})^.i
LDL	87	B	Load Local Word. sp := sp - 1; sp^.i := (mp + mscw_sz - 1 + B)^.i
LLA	84	B	Load Local Address. sp := sp - 1; sp^.p := mp + mscw_sz - 1 + B
STL	A4	B	Store Local Word. (mp + mscw_sz - 1 + B)^.i := sp^.i; sp := sp + 1
			Global One Word Loads and Store
SLDOi	3Ø..3F		Short Load Global Word. sp := sp - 1; sp^.i := (bp + mscw_sz - 1 + i{1..16})^.i
LDO	85	B	Load Global Word. sp := sp - 1; sp^.i := (bp + mscw_sz - 1 + B)^.i
LAO	86	B	Load Global Address. sp := sp - 1; sp^.p := bp + mscw_sz - 1 + B
SRO	A5	B	Store Global Word. (bp + mscw_sz - 1 + B)^.i := sp^.i; sp := sp + 1

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

Mnemonic	Op-Code in Hex		Semantics
			Intermediate One-Word Loads and Store

LOD	89	DB,B	Load Intermediate Word.
			lm := mp;
			for i := 1 to DB do lm := lm^.m.msstat;
			sp := sp - 1; sp^.i := (lm + mscw_sz - 1 + B)^.i

LDA	88	DB,B	Load Intermediate Address.
			lm := mp;
			for i := 1 to DB do lm := lm^.m.msstat;
			sp := sp - 1; sp^.p := lm + mscw_sz - 1 + B

STR	A6	DB,B	Store Intermediate Word.
			lm := mp;
			for i := 1 do DB do lm := lm^.m.msstat;
			(lm + mscw_sz - 1 + B)^.i := sp^.i; sp := sp + 1

			Indirect One-Word Loads and Store
STO	C4		Store Indirect. (sp + 1)^.p^.i := sp^.i; sp := sp + 2
			Extended One-Word Loads and Store

			procedure Raise(err: integer);
			sp := sp -1; sp^.i := err;
			CXG 2,2;
			
			{ All references to ssv are through getsegb }
			function getsegb(segno: integer): memp;
			if segno < 128 then
			getsegb := ssv^.sv[segno]^.segbase
			else
			getsegb := ctp^.t.sibs[segno - 128]^.segbase;

LDE	9A	UB,B	Load Word Extended. sp := sp - 1; sp^.i := (ssv^.sv[UB]^.segbase + B)^.i
LAE	9B	UB,B	Load Address Extended. sp := sp - 1; sp^.p := ssv^.sv[UB]^.segbase + B
STE	D9	UB,B	Store Word Extended. (ssv^.sv[UB]^.segbase + B)^.i := sp^.i; sp := sp + 1

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

Mnemonic	Op-Code in Hex		Semantics
			Multiple Word Loads and Stores (Sets and Reals)

LDC	83	B,UB	Load Multiple Word Constant.
			src := segb + B + UB;
			for i := 1 to UB do (sp - i)^.i := (src - i)^.i;
			sp := sp - UB

LDM	DØ	UB	Load Multiple Words.
			src := sp^.p + UB; sp := sp + 1;
			for i := 1 to UB do (sp - i)^.i := (src - i)^.i;
			sp := sp - UB

STM	8E	UB	Store Multiple Words. dst := (sp^.p + UB)^.p; for i := Ø to UB - 1 do (dst + i)^.i := (sp + i)^.i; sp := sp + UB + 1
			Byte Arrays
LDB	A7		Load Byte. (sp + 1)^.i := (sp + 1)^.b[sp^.i]; sp := sp + 1
STB	C8		Store Byte. (sp + 2)^.b[(sp + 1)^.i] := sp^.i; sp := sp + 3

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

Mnemonic	Op-Code in Hex		Semantics
			Record and Array Indexing and Assignment

MOV	C5	B	Move Words.
			src := sp^.p; dst := (sp + 1)^.p; sp := sp + 2;
			for i := Ø to B - 1 do
			(dst + i)^.i := (src + i)^.i

SINDi	78..7F		Short Index and Load Word. sp^.i := (sp^.p + i{Ø..7})^.i
IND	E6	B	Index and Load Word. sp^.i := (sp^.p + B)^.i
INC	E7	B	Increment Field Pointer. sp^.p := sp^.p + B
IXA	D7	B	Index Array. (sp + 1)^.p := (sp + 1)^.p + sp^.i * B; sp := sp + 1

IXP	D8	UB1,UB2	Index Packed Array.
			var inx: integer;
			inx := sp^.i;
			(sp + 1)^.p := (sp + 1)^.p + inx div UB1;
			sp^.i := UB2; sp := sp - 1;
			sp^.i :=(inx mod UB1)*UB2

LDP	C9		Load A Packed Field. (sp + 2)^.i := (sp + 2)^.i<sp^.i : (sp + 1)^.i>; sp := sp + 2
STP	CA		Store into a packed field. (sp + 3)^.p^.i<(sp + 1)^.i : (sp + 2)^.i> := sp^.i; sp := sp + 4
			Logicals
LAND	A1		Logical AND. (sp + 1)^.bool := (sp + 1)^.i and sp^.i; sp := sp + 1
LOR	AØ		Logical OR. (sp + 1)^.bool := (sp + 1)^.i or sp^.i; sp := sp + 1
LNOT	E5		Logical NOT. sp^.i := not sp^.i

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

Mnemonic	Op-Code in Hex	Semantics
		Logicals (Continued)
BNOT	9F	Boolean NOT. sp^.bool := (not sp^.i) and 1
LEUSW	B4	Compare Unsigned Word <=. (sp + 1)^.bool := ((sp + 1)^.p <= sp^.p); sp := sp + 1
GEUSW	B5	Compare Unsigned Word >=. (sp + 1)^.bool := ((sp + 1)^.p >= sp^.p); sp := sp + 1
		Integers
ABI	EØ	Absolute Value of Integer. sp^.i := Abs (sp^.i)
NGI	E1	Negate Integer. sp^.i := -sp^.i
DUP1	E2	Copy Word. sp := sp - 1; sp^.i := (sp + 1)^.i;
ADI	A2	Add Integers. (sp + 1)^.i := (sp + 1)^.i + sp^.i; sp := sp + 1
SBI	A3	Subtract Integers. (sp + 1)^.i := (sp + 1)^.i - sp^.i; sp := sp + 1
MPI	8C	Multiply Integers. (sp + 1)^.i := (sp + 1)^.i * sp^.i; sp := sp + 1
DVI	8D	Divide Integers. if sp^.i = Ø then Raise (div_by_zero_error); (sp + 1)^.i := (sp + 1)^.i div sp^.i; sp := sp + 1
MODI	8F	Modulo Integers. if sp^.i <= Ø then Raise (mod_by_nonpos_error); (sp + 1)^.i := (sp + 1)^.i mod sp^.i; sp := sp + 1 { -x mod x returns x not Ø }

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

```
---------------------------------------------------------------------------
          Op-Code
Mnemonic  in Hex        Semantics
---------------------------------------------------------------------------
                               Integers (Continued)
                               --------------------

CHK       CB            Check Against Subrange Bounds.
                          if ((sp + 1)^.i <= (sp + 2)^.i) and
                             ((sp + 2)^.i <= sp^.i) then sp := sp + 2
                          else Raise (range_error)

EQUI      B0            Compare Integer =.
                          (sp + 1)^.bool := ((sp + 1)^.i = sp^.i); sp := sp + 1

NEQI      B1            Compare Integer <>.
                          (sp + 1)^.bool := ((sp + 1)^.i <> sp^.i); sp := sp + 1

LEQI      B2            Compare Integer <=.
                          (sp + 1)^.bool := ((sp + 1)^.i <= sp^.i); sp := sp + 1

GEQI      B3            Compare Integer >=.
                          (sp + 1)^.bool := ((sp + 1)^.i >= sp^.i); sp := sp + 1

                                     Reals
                                     -----

{ Over/underflow causes floating-point exception to be raised. }

FLT       CC            Float Top-of-Stack.
                          (sp - real_sz + 1)^.r := Float (sp^.i);
                          sp := sp - real_sz + 1

TNC       BE            Truncate Real.
                          (sp + real_sz - 1)^.i := Truncate (sp^.r);
                          sp := sp + real_sz - 1

RND       BF            Round Real.
                          (sp + real_sz - 1)^.i := Round (sp^.r);
                          sp := sp + real_sz - 1

ABR       E3            Absolute Value of Real.
                          sp^.r := Abs (sp^.r)

NGR       E4            Negate Real.
                          sp^.r := -sp^.r

DUP2      C6            Copy Doubleword.
                          sp := sp - 2;
                          sp^.i := (sp + 2)^.i;  (sp + 1)^.i := (sp + 3)^.i;
---------------------------------------------------------------------------
```

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

Mnemonic	Op-Code in Hex	Semantics
		Reals (Continued)
ADR	C0	Add Reals. (sp + real_sz)^.r := (sp + real_sz)^.r + sp^.r; sp := sp + real_sz
SBR	C1	Subtract Reals. (sp + real_sz)^.r := (sp + real_sz)^.r - sp^.r; sp := sp + real_sz
MPR	C2	Multiply Reals. (sp + real_sz)^.r := (sp + real_sz)^.r * sp^.r; sp := sp + real_sz
DVR	C3	Divide Reals. (sp + real_sz)^.r := (sp + real_sz)^.r / sp^.r; sp := sp + real_sz
EQUREAL	CD	Compare Real =. (sp + 2*real_sz - 1)^.bool := ((sp + real_sz)^.r = sp^.r); sp := sp + 2*real_sz - 1
LEQREAL	CE	Compare Real <=. (sp + 2*real_sz - 1)^.bool := ((sp + real_sz)^.r <= sp^.r); sp := sp + 2*real_sz - 1
GEQREAL	CF	Compare Real >. (sp + 2*real_sz - 1)^.bool := ((sp + real_sz)^.r >= sp^.r); sp := sp + 2*real_sz - 1

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

Mnemonic	Op-Code in Hex		Semantics

Sets

```
ADJ     C7      UB      Adjust Set.
                          var len: integer;
                          len := sp^.i;
                          src := sp + 1; dst := sp + len - UB + 1;
                          if len > UB then { shrink set }
                            for i := 1 to UB do
                              (dst + UB - i).i := (src + UB - i).i
                          else if len < UB then { expand set }
                            for i := Ø to len - 1 do
                              (dst + i)^.i := (src + i)^.i;
                            for i := len to UB - 1 do
                              (dst + i)^.i := Ø;
                          sp := sp + len - UB + 1

SRS     BC              Build Subrange Set.
                          var hi,lo,len: integer;
                          hi := sp^.i; lo := (sp + 1)^.i;
                          if (Ø <= hi) and (hi <= bset_sz-1) and
                             (Ø <= lo) and (lo <= bset_sz-1)
                          then
                            if lo > hi then
                              sp := sp + 1; sp^.i := Ø {Null set}
                            else
                              len := hi div word_sz + 1;
                              sp := sp - len + 1; sp^.i := len;
                              for i := Ø to len * word_sz - 1 do
                                (sp + 1)^.sb[i] := (lo <= i) and (i <= hi);
                          else
                            Raise(range_error)

INN     DA              Set Membership.
                          var len, val: integer;
                          len := sp^.i; val := (sp + len + 1)^.i;
                          if (Ø <= val) and (val <= len * word_sz - 1) then
                            (sp + len + 1)^.bool := (sp + 1)^.sb[val]
                          else (sp + len + 1)^.bool := false;
                          sp := sp + len + 1
```

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

Mnemonic	Op-Code in Hex	Semantics

Sets (Continued)

UNI DB

```
Set Union.
  var lenØ,len1: integer;
  lenØ := sp^.i; len1 := (sp + lenØ + 1)^.i;
  if len1 >= lenØ then {best case move & cut back}
    src := (sp + 1)^.p; dst := (sp + lenØ + 2)^.p;
    for i := Ø to lenØ - 1 do
      (dst + i)^.i := (dst + i)^.i or (src + i)^.i;
    sp := sp + lenØ + 1;
  else
    src := (sp + lenØ + 2)^.p; dst := (sp + 1)^.p;
    for i := Ø to len1 - 1 do
      (dst + i)^.i := (dst + i)^.i or (src + i)^.i;
    { Move set up }
    src := lp + lenØ;  dst := sp + lenØ + len1 + 1;
    for i := Ø to lenØ do { move length word }
      (dst - i)^.i := (src - i)^.i;
    sp := sp + len1 + 1
```

INT DC

```
Set Intersection.
  var lenØ,len1: integer;
  lenØ := sp^.i; len1 := (sp + lenØ + 1)^.i;
  if lenØ = Ø then
    sp := sp + len1 + 1; sp^.i := Ø
  else if len1 = Ø then
    sp := sp + lenØ + 1
  else if len1 > lenØ then {best case move & cut back}
    src := (sp + 1)^.p; dst := (sp + lenØ + 2)^.p;
    for i := Ø to lenØ - 1 do
      (dst + i)^.i := (dst + i)^.i and (src + i)^.i;
    for i := lenØ to len1 - 1 do
      (dst + i)^.i := Ø;
    sp := sp + lenØ + 1;
  else
    dst := (sp + lenØ + 2)^.p; src := (sp + 1)^.p;
    for i := Ø to len1 - 1 do
      (dst + i)^.i := (dst + i)^.i and (src + i)^.i;
    sp := sp + lenØ + 1
```

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

```
---------------------------------------------------------------------------
              Op-Code
Mnemonic   in Hex          Semantics
---------------------------------------------------------------------------
                                 Sets (Continued)
                                 ----------------

DIF        DD              Set Difference.
                             var lenØ,len1: integer;
                             lenØ := sp^.i; len1 := (sp + lenØ + 1)^.i;
                             if lenØ = Ø then
                               sp := sp + 1
                             else if len1 = Ø then
                               sp := sp + lenØ + 1
                             else if len1 > lenØ then {best case move & cut back}
                               src := (sp + 1)^.p; dst := (sp + lenØ + 2)^.p;
                               for i := Ø to lenØ - 1 do
                                 (dst + i)^.i := (dst + i)^.i and not (src + i)^.i;
                               sp := sp + lenØ + 1;
                             else
                               dst := (sp + lenØ + 2)^.p; src := (sp + 1)^.p;
                               for i := Ø to len1 - 1 do
                                 (dst + i)^.i := (dst + i)^.i and not (src + i)^.i;
                               sp := sp + lenØ + 1

EQUPWR     B6              Set Compare =.
                             var lenØ,len1,min1,max1: integer;
                             lenØ := sp^.i; len1 := (sp + lenØ + 1)^.i; i := Ø;
                             min1 := min(lenØ,len1);  max1 := max(lenØ,len1);
                             src := (sp + 1)^.p; dst := (sp + lenØ + 2)^.p;
                             while (i < min1) and ((src + i)^.p = (dst + i)^.p) do
                               i := i + 1;
                             if lenØ > len1 then
                               while (i < max1) and ((src + i)^.p = Ø) do i := i + 1
                             else if len1 > lenØ then
                               while (i < max1) and ((dst + i)^.p = Ø) do i := i + 1;
                             sp := sp + lenØ + len1 + 1; sp^.bool := (i >= max1)

LEQPWR     B7              Set Compare <= (Subset of).
                             var lenØ,len1,min1,max1: integer;
                             lenØ := sp^.i; len1 := (sp + lenØ + 1)^.i; i := Ø;
                             min1 := min(lenØ,len1);  max1 := max(lenØ,len1);
                             src := (sp + 1)^.p; dst := (sp + lenØ + 2)^.p;
                             while (i < min1) and
                                 ((src + i)^.p = (dst + i)^.p or (src + i)^.p) do
                               i := i + 1;
                             if len1 > lenØ then
                               while (i < max1) and ((dst + i)^.p = Ø) do i := i + 1;
                             else i := max1;
                             sp := sp + lenØ + len1 + 1; sp^.bool := (i >= max1)
---------------------------------------------------------------------------
```

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

```
-----------------------------------------------------------------------------
              Op-Code
Mnemonic      in Hex          Semantics
-----------------------------------------------------------------------------
                                    Sets (Continued)
                                    ----------------

GEQPWR    B8                  Set Compare >= (Superset of).
                                var lenØ,len1,min1,max1: integer;
                                lenØ := sp^.i; len1 := (sp + lenØ + 1)^.i; i := Ø;
                                min1 := min(lenØ,len1);  max1 := max(lenØ,len1);
                                src := (sp + 1)^.p; dst := (sp + lenØ + 2)^.p;
                                while (i < min1) and
                                    ((dst + i)^.p = (dst + i)^.p or (src + i)^.p) do
                                  i := i + 1;
                                if len1 < lenØ then
                                  while (i < max1) and ((src + i)^.p = Ø) do i := i + 1;
                                else i := max1;
                                sp := sp + lenØ + len1 + 1; sp^.bool := (i >= max1)

                                      Byte Arrays
                                      -----------

EQUBYT    B9        B         Equal Byte Array Compare.
                                src := sp^.p;  dst := (sp + 1)^.p;  i := Ø;
                                while (i < B) and (src^.b[i] = dst^.b[i]) do
                                  i := i + 1;
                                sp := sp + 1;  sp^.bool := (i >= B)

LEQBYT    BA        B         Less or Equal Byte Array Compare.
                                src := sp^.p;  dst := (sp + 1)^.p;  i := Ø;
                                while (i < B) and (src^.b[i] <= dst^.b[i]) do
                                  i := i + 1;
                                sp := sp + 1;  sp^.bool := (i >= B)

GEQBYT    BB        B         Greater or Equal Byte Array Compare.
                                src := sp^.p;  dst := (sp + 1)^.p;  i := Ø;
                                while (i < B) and (src^.b[i] >= dst^.b[i])
                                  i := i + 1;
                                sp := sp + 1;  sp^.bool := (i >= B)

                                         Jumps
                                         -----

UJP       8A        SB        Unconditional Jump.
                                pc := pc + SB

FJP       D4        SB        False Jump.
                                if not sp^.bool then pc := pc + SB;  sp := sp + 1

EFJ       D2        SB        Equal False Jump.
                                if (sp + 1)^.i <> sp^.i then pc := pc + SB;
                                sp := sp + 2
-----------------------------------------------------------------------------
```

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

```
--------------------------------------------------------------------------
            Op-Code
Mnemonic    in Hex              Semantics
--------------------------------------------------------------------------
                                     Jumps (Continued)
                                     -----------------

NFJ       D3      SB       Not Equal False Jump.
                             if (sp + 1)^.i = sp^.i then pc := pc + SB;
                             sp := sp + 2

UJPL      8B      W        Unconditional Long Jump.
                             pc := pc + W

FJPL      D5      W        False Long Jump.
                             if not sp^.bool then pc := pc + W;  sp := sp + 1

XJP       D6      B        Case Jump.
                             if ((segb + B)^.i <= sp^.i) and
                                  ((segb + B + 1)^.i >= sp^.i) then
                               pc := pc + (segb + B + 2 + sp^.i - (segb + B)^.i)^.p;
                             sp := sp + 1

                       Procedure and Function Calls and Returns
                       ----------------------------------------

                           procedure createmscw;
                             { data_sz =  (segb + segb^.i - procno)^.i }
                             sp := sp - mscw_sz - data_sz;
                             if (sp < splow) or (data_sz + mscw_sz > sp - splow
                                then Raise(stack_overflow);
                             lm := mp; mp := sp;
                             mp^.m.msdynl := lm; mp^.m.msipc := pc

CPL       90      UB       Call Local Procedure.
                             createmscw;
                             mp^.m.msstat := lm; mp^.m.msseg := 0;
                             pc := (segb + segb^.i - UB - 1)^.p

CPG       91      UB       Call Global Procedure.
                             createmscw;
                             mp^.m.msstat := bp; mp^.m.msseg := 0;
                             pc := (segb + segb^.i - UB - 1)^.p

CPI       92      DB,UB    Call Intermediate Procedure.
                             createmscw;
                             mp^.m.msseg := 0;
                             lm := mp;
                             for i := 1 to DB do lm := lm^.m.msstat;
                             mp^.m.msstat := lm;
                             pc := (segb + segb^.i - UB - 1)^.p
--------------------------------------------------------------------------
```

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

Mnemonic	Op-Code in Hex		Semantics
			Procedure and Function Calls and Returns (Continued)

CXL	93	UB1,UB2	Call Local External Procedure.
			createmscw;
			mp^.m.msstat := lm;
			mp^.m.msseg := (segb + segb^.i)^.b[Ø];
			segb := ssv^.sv[UB1]^.segbase;
			ssv^.sv[UB1]^.segrefs := ssv^.sv[UB1]^.segrefs + 1;
			pc := (segb + segb^.i - UB - 1)^.p

CXG	94	UB1,UB2	Call Global External Procedure.
			createmscw;
			mp^.m.msstat := bp;
			mp^.m.msseg := (segb + segb^.i)^.b[Ø];
			segb := ssv^.sv[UB1]^.segbase;
			ssv^.sv[UB1]^.segrefs := ssv^.sv[UB1]^.segrefs + 1;
			pc := (segb + segb^.i - UB - 1)^.p

CXI	95	UB1,DB,UB2	Call Intermediate External Procedure.
			createmscw;
			lm := mp;
			for i := 1 to DB do lm := lm^.m.msstat;
			mp^.m.msstat := lm;
			mp^.msseg := (segb + segb^.i)^.b[Ø];
			segb := ssv^.sv[UB1]^.segbase;
			ssv^.sv[UB1]^.segrefs := ssv^.sv[UB1]^.segrefs + 1;
			pc := (segb + segb^.i - UB - 1)^.p

CPF	97		Call Formal Procedure.
			var ls: memtrix;
			ls := sp^.i; lm := (sp + 1)^.p; sp := sp + 2;
			createmscw;
			mp^.m.msseg := ls.b[1];
			segb := ssv^.sv[ls.b[1]]^.segbase;
			ssv^.sv[ls.b[1]]^.segrefs :=
			  ssv^.sv[ls.b[1]]^.segrefs + 1;
			mp^.m.msstat := lm;
			pc := (segb + segb^.i - ls.b[Ø] - 1)^.p

RPU	96	B	Return From User Procedure.
			sp := mp; lm := mp;
			mp := lm^.m.msdynl; pc := lm^.m.msipc;
			if lm^.m.msseg <> Ø then
			  segb := ssv^.sv[lm^.m.msseg].segbase;
			  ssv^.sv[m.msseg]^.segrefs :=
			    ssv^.sv[m.msseg]^.segrefs - 1;
			sp := sp + B + mscw_sz

Mnemonic	Op-Code in Hex	Semantics

Procedure and Function Calls and Returns (Continued)

LSL 99 DB Load Static Link Onto Stack.

```
  lm := mp;
  for i := 1 to DB do lm := lm^.m.msstat;
  sp := sp - 1; sp^.p := lm
```

Concurrency Control

```
var qhead, qtask: tibp;
procedure updatetib;
  ctp^.t.mp := mp; ctp^.t.bp := bp;
  ctp^.t.sp := sp; ctp^.t.ipc := pc;
  ctp^.t.segb := segb;

procedure enque;
  var tl,t2: tibp;
  tl := qhead; t2 := NIL;
  while tl <> NIL do
    if tl^.prior < qtask^.prior then goto 1;
    t2 := tl; tl := tl^.qlink;
1:
  qtask.qlink := tl;
  if t2 = NIL then qhead := qtask
    else t2^.qlink := qtask

procedure deque;
  qtask := qhead; qhead := qhead^.qlink;

5: { taskswitch }
  updatetib;
6:
  while rq = NIL do
    if an_interrupt then
      ctp := NIL;
      sp := sp - 1;
      sp^.i := int_vec_address; {hardware generated}
      sp^.p := int_vec_address^.p;
      goto SIGNAL;
  qhead := rq; deque;
  rq := qhead; ctp := qtask;
  sp := ctp^.t.sp; mp := ctp^.t.mp; bp := ctp^.t.bp;
  pc := ctp^.t.pc; segb := ctp^.t.segb;
  { Fall through here as well as for all other
     operators implies fetch next instruction }
```

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

Mnemonic	Op-Code in Hex	Semantics
		Concurrency Control (Continued)

SIGNAL	DE	Signal semaphore.
		`if sp^.p^.sem.count = Ø then`
		`  if sp^.p^.sem.waitq <> NIL then`
		`    qhead := sp^.p^.sem.waitq; deque;`
		`    sp^.p^.sem.waitq := qhead; qhead := rq;`
		`    enque; rq := qhead;`
		`    if ctp = nil then goto 6;`
		`    if ctp^.prior < qtask^.prior then`
		`      qtask := ctp; qhead := rq; enque;`
		`      rq := qhead;`
		`      goto 5;`
		`    else goto 3;`
		`sp^.p^.sem.count := sp^.p^.sem.count + 1`
		`if ctp = NIL then goto 6;`
		`3:`
		`sp := sp + 1`

WAIT	DF	Wait on Semaphore.
		`if sp^.p^.sem.count = Ø then`
		`  qhead := sp^.p^.sem.waitq;`
		`  qtask := ctp; enque;`
		`  sp^.p^.sem.waitq := qhead;`
		`  goto 5;`
		`else sp^.p^.sem.count := sp^.p^.sem.count + 1;`
		`sp := sp + 1`

Table C-3. P-Code Operators in a Pascal-like Metalanguage. (Continued)

Mnemonic	Op-Code in Hex	Semantics
		Miscellaneous
LPR	9D	Load Processor Register. if sp^.i >= Ø then updatetib; sp^.i := case sp^.i of -3: rq; -2: ssv; -1: ctp; 1..maxint: (ctp + sp^.i)^.i
SPR	D1	Store Processor Register. if sp^.i >= -1 then updatetib; sp^.p^.i := case sp^.i of -3: rq; -2: ssv; -1: ctp; goto 5; { Taskswitch } 1..maxint: (ctp + (sp + 1)^.i)^.i if sp^.i >= -1 then updatetib; sp := sp + 2
BPT	9E	Break Point. Raise(Breakpoint);
NOP	9C	No Operation.

SWAP	BD	Swap Word.
		i := sp^.i;
		sp^.i := (sp + 1)^.i;
		(sp + 1)^.i := i

APPENDIX D. AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE

Version 3.Ø

	Oct	Hex	Char		Oct	Hex	Char		Oct	Hex	Char		Oct	Hex	Char
Ø	ØØØ	ØØ	NUL	32	Ø4Ø	2Ø	SP	64	1ØØ	4Ø	@	96	14Ø	6Ø	`
1	ØØ1	Ø1	SOH	33	Ø4Ø	21	!	65	1Ø1	41	A	97	141	64	a
2	ØØ2	Ø2	STX	34	Ø42	22	"	66	1Ø2	42	B	98	142	62	b
3	ØØ3	Ø3	ETX	35	Ø43	23	#	67	1Ø3	43	C	99	143	63	c
4	ØØ4	Ø4	EOT	36	Ø44	24	$	78	1Ø4	44	D	1ØØ	144	64	d
5	ØØ5	Ø5	ENG	37	Ø45	25	%	69	1Ø5	45	E	1Ø1	145	65	e
6	ØØ6	Ø6	ACK	38	Ø46	26	&	7Ø	1Ø6	46	F	1Ø2	146	66	f
7	ØØ7	Ø7	BEL	39	Ø47	27	'	71	1Ø7	47	G	1Ø3	147	67	g
8	Ø1Ø	Ø8	BS	4Ø	Ø5Ø	28	(	72	11Ø	48	H	1Ø4	15Ø	68	h
9	Ø11	Ø9	HT	41	Ø51	29	)	73	111	49	I	1Ø5	151	69	i
1Ø	Ø12	ØA	LF	42	Ø52	2A	*	74	112	4A	J	1Ø6	152	6A	j
11	Ø13	ØB	VT	43	Ø53	2B	+	75	113	4B	K	1Ø7	153	6B	k
12	Ø14	ØC	FF	44	Ø54	2C	,	76	114	4C	L	1Ø8	154	6C	l
13	Ø15	ØD	CR	45	Ø55	2D	-	77	115	4D	M	1Ø9	155	6D	m
14	Ø16	ØE	SO	46	Ø56	2E	.	78	116	4E	N	11Ø	156	6E	n
15	Ø17	ØF	SI	47	Ø57	2F	/	79	117	4F	O	111	157	6F	o
16	Ø2Ø	1Ø	DLE	48	Ø6Ø	3Ø	Ø	8Ø	12Ø	5Ø	P	112	16Ø	7Ø	p
17	Ø21	11	DC1	49	Ø61	31	1	81	121	51	Q	113	161	71	q
18	Ø22	12	DC2	5Ø	Ø62	32	2	82	122	52	R	114	162	72	r
19	Ø23	13	DC3	51	Ø63	33	3	83	123	53	S	115	163	73	s
2Ø	Ø24	14	DC4	52	Ø64	34	4	84	124	54	T	116	164	74	t
21	Ø25	15	NAK	53	Ø64	35	5	85	125	55	U	117	165	75	u
22	Ø26	16	SYN	54	Ø66	36	6	86	126	56	V	118	166	76	v
23	Ø27	17	ETB	55	Ø67	37	7	87	127	57	W	119	167	77	w
24	Ø3Ø	18	CAN	56	Ø7Ø	38	8	89	13Ø	58	X	12Ø	17Ø	78	x
25	Ø31	19	EM	57	Ø71	39	9	89	131	59	Y	121	171	79	y
26	Ø32	1A	SUB	58	Ø72	3A	:	9Ø	132	5A	Z	122	172	7A	z
27	Ø33	1B	ESC	59	Ø73	3B	;	91	133	5B	[	123	173	7B	{
28	Ø34	1C	FS	6Ø	Ø74	3C	<	92	134	5C	\	124	174	7C	|
29	Ø35	1D	GS	61	Ø75	3D	=	93	135	5D	]	125	175	7D	}
3Ø	Ø36	1E	RS	62	Ø76	3E	>	94	136	5E	^	126	176	7E	~
31	3Ø7	1F	US	63	Ø77	3F	?	95	137	5F	_	127	177	7F	DEL

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

APPENDIX E. UCSD PASCAL RESERVED WORDS

Version 3.Ø

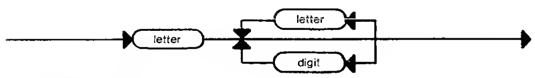
AND	GOTO	RECORD
ARRAY		REPEAT
	IF	
BEGIN	IMPLEMENTATION	SET
	IN	SEGMENT
CASE	INTERFACE	SEPARATE
CONST		
	LABEL	THEN
DIV		TO
DO	MOD	TYPE
DOWNTO		
	NOT	UNIT
ELSE		UNTIL
END	OF	USES
	OR	
FILE		VAR
FOR	PACKED	
FORWARD	PROCEDURE	WHILE
FUNCTION	PROCESS	WITH
	PROGRAM	

A syntax error results if an attempt is made to declare a reserved word as an identifier.

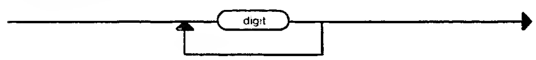
THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

APPENDIX F. UCSD PASCAL SYNTAX DIAGRAMS

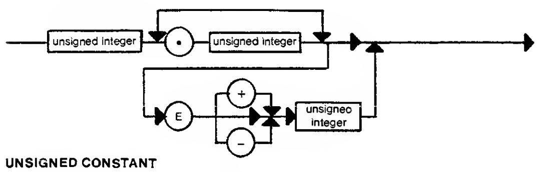
IDENTIFIER



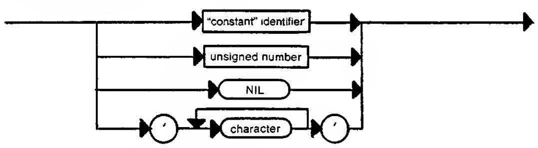
UNSIGNED INTEGER



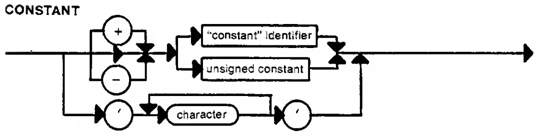
UNSIGNED NUMBER



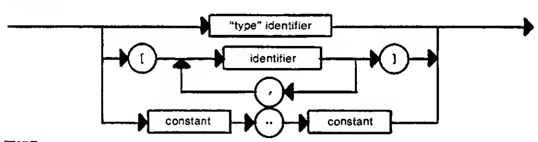
UNSIGNED CONSTANT



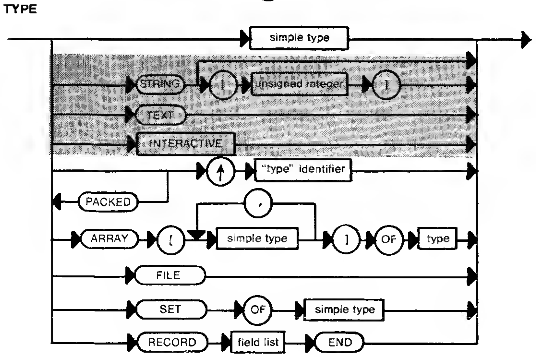
CONSTANT



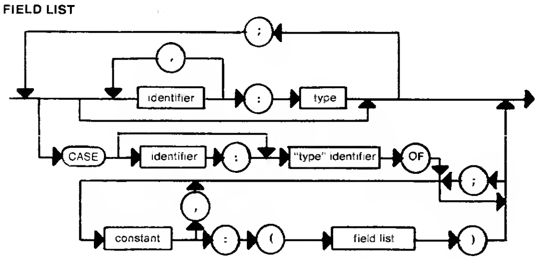
SIMPLE TYPE



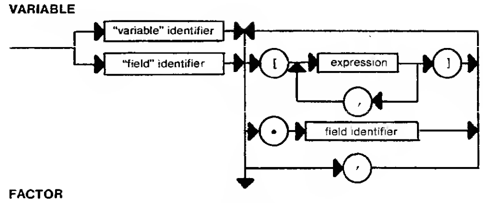
TYPE



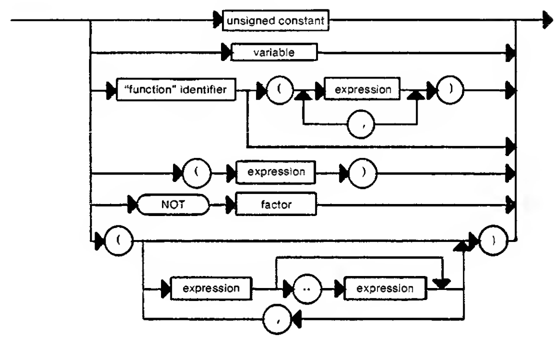
FIELD LIST



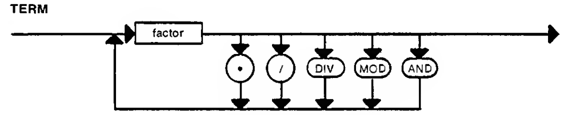
VARIABLE



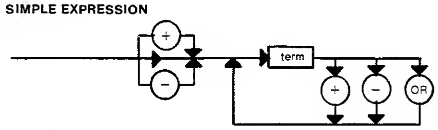
FACTOR



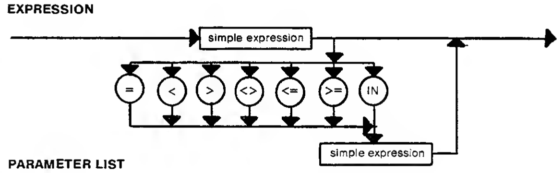
TERM



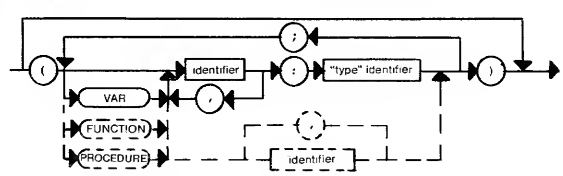
SIMPLE EXPRESSION



EXPRESSION

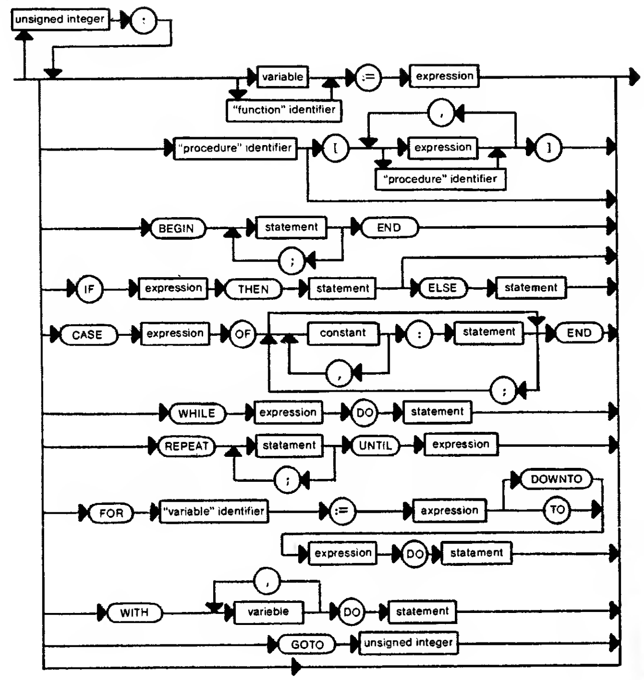


PARAMETER LIST



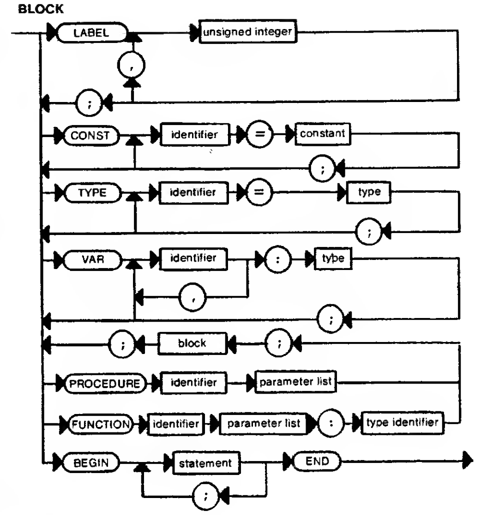
UCSD PASCAL SYNTAX DIAGRAMS CONTINUED

STATEMENT

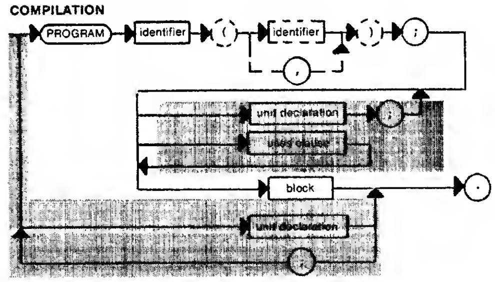


DASHED LINES (--) ARE NOT INCLUDED
SHADED AREAS REPRESENT UCSD EXTENSIONS

BLOCK



COMPILATION



APPENDIX G: ME1600 AND SB1600 I/O ADDRESSES

This appendix presents several tables of I/O addresses for the ME and SB1600 series machines.

Table G-1. ME1600 I/O Addresses.

Hex	Decimal	Function
FC10	-1008	Serial port A (unit #1:,#2:)
FC15	-1003	Interrupt base register serial ports A-D, Par port #6:
FC16	-1002	Serial port B (unit #7:,#8:)
FC1C	-996	Serial port C (unit #15:,#16:)
FC22	-990	Serial port D (unit #17:,#18:)
FC28	-984	Parallel port (unit #6:)
FC2C	-980	Interrupt mask register serial ports A-D, Par port #6:
FC30	-976	Floppy port (unit #4:,#5:,#9:,#10:)
FC34	-972	DMA EOB and DINTR
FC35	-971	Floppy interrupt priority
FC36	-970	Floppy interrupt base register
FC40	-960	Microcode use during interrupt handling
FC41	-959	Interrupt enable address
FC42	-958	Interrupt mask register for RTC,BTO,PFD, clock tick rate
FC43	-957	Interrupt base register for RTC,BTO,PFD
FC60	-928	Microcode use during interrupt handling
FC68	-922	Used by microcode to determine boot from ROM
FC70	-912	Winchester disk
FD10	-752	Serial port E (unit #19:,#20:)
FD15	-747	Interrupt base register serial ports E-H, par port #27:
FD16	-746	Serial port F (unit #21:,#22:)
FD1C	-740	Serial port G (unit #23:,#24:)
FD22	-734	Serial port H (unit #25:,#26:)
FD28	-728	Parallel port (unit #27:)
FD2C	-724	Interrupt mask register serial ports E-H, par port #27:
FD30	-720	Floppy port (unit #11:,#12:,#13:,#14:)
FD34	-716	DMA EOB and DINTR
FD35	-715	Floppy interrupt priority
FD36	-714	Floppy interrupt base register
FF00 to FFFF		ROM address space

Table G-2. Interrupt Addresses.

Hex	Decimal	Function
0010	16	PFD Power fail detect
0011	17	BTO Bus time out
0012	18	RTC Real time clock
0016	22	Winchester disk
001E	30	Floppy (unit #4:,#5:,#9:,#10:)
001F	31	Floppy (unit #11:,#12:,#13:,#14:)
0020	32	--- not used ---
0021	33	--- not used ---
0022	34	Serial port D output buffer empty
0023	35	Serial port D input buffer full
0024	36	Serial port D exception
0025	37	Serial port C output buffer empty
0026	38	Serial port C input buffer full
0027	39	Serial port C exception
0028	40	Parallel port #6: output
0029	41	Serial port B output buffer empty
002A	42	Serial port B input buffer full
002B	43	Serial port B exception
002C	44	Parallel port #6: input
002D	45	Serial port A output buffer empty
002E	46	Serial port A input buffer full
002F	47	Serial port A exception
0030	48	--- not used ---
0031	49	--- not used ---
0032	50	Serial port H output buffer empty
0033	51	Serial port H input buffer full
0034	52	Serial port H exception
0035	53	Serial port G output buffer empty
0036	54	Serial port G input buffer full
0037	55	Serial port G exception
0038	56	Parallel port #27: output
0039	57	Serial port F output buffer empty
003A	58	Serial port F input buffer full
003B	59	Serial port F exception
003C	60	Parallel port #27: input
003D	61	Serial port E output buffer empty
003E	62	Serial port E input buffer full
003F	63	Serial port E exception

Table G-3. Mask Registers.

Hex	Decimal	Function
FC2C	-980	Serial ports A-D, parallel port #6:

Bit			
15 Port A exception	11 Port B exception	7 Port C exception	3 Port D input
14 Port A input	10 Port B input	6 Port C input	2 Port D output
13 Port A output	9 Port B output	5 Port C output	1 unused
12 Parallel #6 input	8 Parallel #6 output	4 Port D exception	0 unused

Hex	Decimal	Function
FD2C	-724	Serial ports E-H, parallel port #27:

Bit			
15 Port E exception	11 Port F exception	7 Port G exception	3 Port H input
14 Port E input	10 Port F input	6 Port G input	2 Port H output
13 Port E output	9 Port F output	5 Port G output	1 unused
12 Parallel #6 input	8 Parallel #6 output	4 Port H exception	0 unused

Hex	Decimal	Function
FC42	-958	BTO,PFD,RTC and clockvalue mask

Bit			
5	BTP	3,2,1	clock rates
4	PFD	0	RTC

|NOTE|

Each mask bit set to 1 means the corresponding interrupt is enabled.

Table G-4. SB1600 I/O Addresses.

Hex	Decimal	Function
FC10	-1008	Serial port A (unit #1:,#2:)
FC18	-1000	System status word/ system control word
FC20	-992	Serial port B (unit #7:,#8:)
FC30	-976	Floppy port (unit #4:,#5:,#9:,#10:)
FC40	-960	Microcode use during interrupt handling
FC42	-958	Interrupt mask register for RTC,BTO,PFD
FC43	-957	Interrupt base register for RTC,BTO,PFD
FC48	-952	Interrupt enable address
FC4C	-948	Reserved for DES, TOD
FC60	-928	Microcode use during interrupt handling
FC68	-922	Used by microcode to determine boot from ROM/ Density sel
FC6C	-918	Parity error address / Disable parity check
FC70	-912	Parallel port #6:
FE00 to FFFF		ROM address space

Table G-5. SB1600 Interrupt Addresses.

Hex	Decimal	Function	
0000	00	BTO Bus time out, Parity error, Interrupt reply time out	
0020	32	Floppy (unit #4:,#5:,#9:,#10:)	
0024	36	Serial port A input buffer full	22832
0028	40	Serial port B output buffer empty	(22[illegible]
002C	44	Serial port B input buffer full	(22[illegible]
0030	48	Serial port A output buffer empty	(22[illegible]
0034	52	Serial port A, Serial port B exception	(22[illegible]
0038	56	Parallel port #6: input	
003C	60	Parallel port #6: output	

Table G-6. SB1600 Control Register.

Hex	Decimal	Function

FC18 is the System Control Register/System Status Register depending whether it is read or written.

FC18 read mode

Bits	Function
15	Set to 1 to identify G Board status
14..10	Unused
9	Set when parity error
8	Unused
7	Set when memory reply time out
6	Set when interrupts enabled
5	Set when EOB true for DMA
4	Set when DINTR for DMA
3	Set when double density enabled
2	Set when booting from floppy disk
1	Set when 8 inch floppies in use, reset for 5.25 inch floppies
0	Set when interrupt reply time out

FC18 write mode

Bits	Function
15..12	Diagnostic Result Bits (hardware test points)
11..8	Baudrate set for serial port B
7..0	Unused

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

APPENDIX H: BOOT AND INITIALIZATION DIAGNOSTIC MESSAGES

The H3 boot and operating system initialization sequence provide information to the user as to the bootstrap process. This information proves valuable in the event that the system does not boot. This information is in the form of words or numbers displayed on the screen at various stages of the booting process. In a properly working system, this display quickly changes, showing the progress, until finally the display is cleared, and the system prompt line and welcome message appear. In the event of failure on the part of either the software or the hardware, the display stops at the point the problem occurred. The current state of the display could be given to a service representative and appropriate action may be taken.

Four main actions occur in system booting and initialization. The first is execution of the PROM program (or microcode in the absence of PROM) which determines from which disk to boot; then a small section of code called the primary bootstrap is loaded.

On SB1600 series machines, the PROM also performs low-level hardware diagnostics. On machines with older PROMs or no PROMs, no display occurs at this point.

Two types of PROMs are used: one for the SB1600 series machines and one for the ME1600 series machines. The SB1600 PROM first performs hardware diagnostics; then it locates the first floppy drive with a diskette and loads the primary boot from that diskette. The ME1600 PROM does not perform diagnostics but does look for a floppy drive with a diskette in it. If no floppy drive is found containing a diskette, a Winchester drive is sought from which to boot.

H.1 SB1600 PROM

The SB1600 PROM normally displays: TESTING 1.23. Normal deviations from this display include a repeating period before the "1" until a drive with a diskette in it is found. Also, a second period may appear after the "1" if a diskette is found that is not the same density as that originally set in the hardware. Any other deviations strongly suggest hardware problems and should be pointed out to a service representative.

The following subsections describe the details of the SB1600 PROM boot, built-in-test, and error codes.

H.1.1 Built-In-Test (BIT)

DRB set to 0000 at Master Reset. (The DRB, diagnostic result bits, consist of four hardware probe points.)

DRB set to 1111 at first instruction in PROM. No RAM is needed at this step.

Setup pointers in memory to memory mapped I/O space and fill all memory with 0000. DRB set to 0001. Carriage return written to CRT. Line feed written to CRT.

Write TIB to 7E00. Test high memory (8000 to FBFF) by writing FFFF and then reading. If a test problem, error code 'a' written to CRT. 'T' written to CRT.

Relocate stack to high memory. 'E' written to CRT. Transfer control to high memory. 'S' written to CRT.

Test low memory (0000 to 7FFF) by writing FFFF and then reading. If a test problem, error code 'b' written to CRT. 'T' written to CRT.

Test for memory reply time out, interrupts disabled, interrupt time out. If a test problem, error code 'c' written to CRT. 'I' written to CRT.

Test DMA register write/read. If a test problem, error code 'd' written to CRT. 'N' written to CRT.

Test Floppy register write/read. If a test problem, error code 'e' written to CRT. 'G' written to CRT.

Test serial port B register write/read. If a test problem, error code 'f' written to CRT.

H.1.2 Boot from Floppy Disk

' ' written to CRT. Floppy given a force interrupt command and then drive ready status interrogated. Command sequencing is drive 0,1,2,3 corresponding to drives 4,5,9,10. '.' written to CRT for each not ready drive.

'l' written to CRT if a ready drive found.

Issue restore to track 0 command to ready drive. If a problem, error code 'h' written to CRT. DRB set to 0010.

'.' written to CRT. Issue seek to track 1 command to ready drive. DRB set to 0011. If seek failure occurs, switch density and try until a disk can be read.

'2' written to CRT. Issue read track 1 command to ready drive. If a problem, error code 'i' written to CRT.

DRB set to Ø1ØØ. Check read error status. If no error, then '3' written to CRT.

Task switch to software boot that has just been read in from floppy.

H.1.3 Error Codes

'a' Failure in high memory test.

'b' Failure in low memory test.

'c' Memory reply timeout failure or interrupt reply timeout failure or interr- upts incorrectly enabled.

'd' DMA register test failure.

'e' Floppy register test failure.

'f' Port B register test failure.

'h' Floppy restore command failure.

'i' Failure to seek track in either density.

'j' Track 1 read error either CRC error, lost data, record not found or DMA time out.

H.2 ME1600 PROM

The ME1600 PROM normally displays: <f>. The "<" is output immediately after the PROM boot begins execution. The "f" signifies that a floppy disk controller board is present. A "w" in place of the "f" signifies that a Winchester disk controller board is present. The ">" means that a successful disk read has completed.

Normal deviations from this display include a repeating "f" until the PROM boot finds a drive containing a diskette or a repeating "fw" pattern if the machine has a Winchester disk controller. The "f" or "fw" continues until the PROM boot finds either a drive containing a floppy disk or a Winchester disk.

Once the primary boot is loaded by PROM or microcode and executed, the primary boot attempts to read the secondary bootstrap. If the secondary boot is being loaded from a floppy, the letters "floppy" are displayed. If the secondary boot is being loaded from a Winchester disk, the letters "winch" are displayed. If none of the messages appear or are not complete (such as "flo"), the disk does not contain a valid boot on it, or disk transfer problems may exist.

Once the secondary boot is loaded from a disk, it displays a message giving the memory size it assumes while loading the operating system. This memory size is a parameter supplied during the bootmake operation. Systems called "128K-byte" system actually have only 126K bytes of RAM that the system can use; the remaining 2K bytes are used for the PROM and hardware device communication. After the memory size is displayed, the secondary boot attempts to load the portions of the operating system necessary to initialize the system.

At that point, control goes to the initialization code of the operating system to set up the I/O driver tasks, system files, and other operating system control structures. During initialization, numbers are written to the screen on top of one another. A few of these numbers are key numbers used to diagnose problems.

If no numbers are displayed after the secondary boot displays its message, the secondary boot most likely failed to load all required portions of the operating system. Below is a list of stopping numbers and the problems they suggest:

6	:	serial or parallel ports
7	:	floppy
8	:	winchester
10	:	serial ports
11..14	:	interrupts
15,16	:	system configuration / unitallocation / winchester
17	:	clock / reading from system disk
0..255c	:	any number followed by a 'c' indicates problems with clearing that unit.
21	:	memory parity
22..27	:	system file initialization
28	:	system not loaded properly / disk reading problem.

After all numbers have been displayed, the welcome message should appear, followed by the prompt line at the top of the screen.

APPENDIX I. III.O OPERATING SYSTEM GLOBALS [H3]

The following code represents the globals for release H3 of the III.Ø Operating System. The order of VAR declarations is stable and is guaranteed not to change from the declaration of SYSCOM to the declaration for UNITABLE.

```
const
   {                constants for pmachine statements                      }
   { Tib Registers        Execute errors             Operators             }

     rqreg    = -3;       syserr =  Ø;         sto   = 196;     leusw = 18Ø;
     ssvreg   = -2;       invinx =  1;         ldm   = 2Ø8;     geusw = 181;
     ctpreg   = -1;       noproc =  3;         ldb   = 167;     adi   = 162;
     priorreg =  1;       stkovr =  4;         stb   = 2ØØ;     sbi   = 163;
     splowreg =  2;       syioer =  9;         mov   = 197;     cxg   = 148;
     spuprreg =  3;       uioerr = 1Ø;         sindØ = 12Ø;     cxi   = 149;
     spreg    =  4;       fperr  = 12;         sind1 = 121;     cpf   = 151;
     mpreg    =  5;       s2long = 13;         inc   = 231;     rpu   = 15Ø;
     bpreg    =  6;                            bnot  = 159;     lsl   = 153;
     usvreg   = 11;                            ixa   = 215;     lpr   = 157;
                                               bpt   = 158;     spr   = 2Ø9;

{
                Definitions common to more than one set of types
}
const
  vidleng       = 7;
  tidleng       = 15;
  maxdir      =    77; { max number of entries in a directory }
  fblksize    =   512; { standard disk block length }
  dirblk      =     2; { disk addr of directory }
  agelimit    =   3ØØ; { max age for gdirp...in ticks (5 seconds) }

  cmaxunit      = 255;
  oldmaxunit    = 27;    { the old (pre-H3) maximum unit number }
  maxsysunit =   127; { maximum number of system serial & parallel units }
  mindiskunit =    4;

  configversion = 'WDØ2';
```

```
{ These are hardware determined numbers }
  maxdrive      = 3;    { 4 possible : Ø..3 }
  { for the Winchester controller }
  maxheads      = 8;    { the maximum number of heads }
  maxcylinder   = 1Ø23; { 1Ø24 possible : Ø..1Ø23 }

type
  vid = string[vidleng];
  tid = string[tidleng];
  alpha = packed array [Ø..7] of char;
  window = packed array [Ø..Ø] of char;
  windowp = ^window;
  dp_integer = record
                 lo : integer;
                 hi : integer;
               end;
```

```
{
                              Unitable related types

                                 Needs t_common
}

    unitnum    = Ø..oldmaxunit;

    devtype = (invalid, floppydisk, parallel, serial, winchdisk {& others});
    unitentry = packed record {an entry of unitable}
                 uvid    : vid;      { VOLUME ID FOR UNIT }
                 uisblkd : boolean;
                 case utype : devtype of
                   floppydisk, winchdisk : (ueovblk: integer);
                   serial, parallel      : (portfor: integer)
                end { unitentry } ;

    arrayunitable = array[Ø..cmaxunit] of ^unitentry;
```

```
{
                        Directory related types
}
                                        { ARCHIVAL INFO...THE DATE }

    daterec = packed record
                month: Ø..12;   { Ø IMPLIES DATE NOT MEANINGFUL }
                day:   Ø..31;   { DAY OF MONTH }
                year:  Ø..1ØØ   { 1ØØ IS TEMP DISK FLAG }
              end { DATEREC } ;

                                        { DISK DIRECTORIES }
    dirrange = Ø..maxdir;

    filekind = (untypedfile, xdskfile, codefile, textfile, infofile,
                datafile, graffile, fotofile, securedir);

    direntry = record
                 dfirstblk: integer; { FIRST PHYSICAL DISK ADDR }
                 dlastblk: integer;  { POINTS AT BLOCK FOLLOWING }
                 case dfkind: filekind of
                   securedir,
                   untypedfile: { ONLY IN DIR[Ø]...VOLUME INFO }
                      (dvid: vid;           { NAME OF DISK VOLUME }
                       deovblk: integer;     { LASTBLK OF VOLUME }
                       dnumfiles: dirrange;  { NUM FILES IN DIR }
                       dloadtime: integer;   { TIME OF LAST ACCESS }
                       dlastboot: daterec);  { MOST RECENT DATE SETTING }
                   xdskfile,codefile,textfile,infofile,
                   datafile,graffile,fotofile:
                      (dtid: tid;              { TITLE OF FILE }
                       dlastbyte: 1..fblksize; { NUM BYTES IN LAST BLOCK }
                       daccess: daterec)       { LAST MODIFICATION DATE }
               end { DIRENTRY } ;

    dirp = ^directory;

    directory = array [dirrange] of direntry;
```

```
{
                    Configuration record related types
}

    { declarations needed for the configuration table }
    floppytype = (eight_inch, five_inch, {others}f2,f3,f4,f5,f6,f7,f8); {4 bits}
    driverange = Ø..maxdrive;
    cylndrange = Ø..maxcylinder;
    diskunits  = mindiskunit..maxunit;
    sysunits   = 1..maxsysunit;
    sunitset   = set of sysunits;            { 8 words}
    unitset    = set of diskunits;           {16 words}

    { system configuration of unit number mapping to types of disk drives }

    punitinfo = ^unitinfo;
    unitinfo  = record
                  cylinder : cylndrange;
                  block    : integer;
                  vollen   : integer;
                end;

    configrec = record
                  version    : packed array[Ø..3] of char;
                          { identifies this as a valid configrec }
                          { this field is initialized when loaded }

                  drive      : packed array [diskunits] of driverange;
                          { the disk controller drive number of unit }

                  { characteristics of a particular Winchester drive }
                  winchdrive : array [driverange] of
                                 record {this record unpacked for speed}
                                   maxcyl         : cylndrange;
                                   numofheads     : Ø..maxheads;
                                   blockspertrack : Ø..255;
                                   step_rate      : Ø..15;
                                 end;

                  { map of unitnumbers to drive and disk location & length }
                  pwinchunit : array [diskunits] of punitinfo;

                end; {configrec}
```

```
static_configrec = { the form of the configuration record on disk }
          record
            version   : packed array[Ø..3] of char;       {2 words}
                    { identifies this as a valid configrec }

            serialset : sunitset;                          {8 words}
                    { the system defined serial units }
            parallset : sunitset;                          {8 words}
                    { the system defined parallel units }
            floppyset : unitset;                           {16 words}
                    { those units that are on floppy drives }

            { this field is currently unused }
            floppydrive : packed array [Ø..7] of floppytype; {2 words}
                    { type of floppy drive }

            { remainder of the 64 words in the last sector on a floppy
              track, for added future data fields }
            reserved  : array [1..28] of integer;

            {------ fields below this line are only on Winchesters -----}

            winchset  : unitset;
                    { those units that are on Winchester drives }

            { map of unitnumbers to drive and disk location & length }
            winchunit  : array [diskunits] of unitinfo;

            { the form the configuration record will take in memory }
            dynamic_config : configrec;

          end; {static_configrec}
```

```
static_configrec = { the form of the configuration record on disk }
          record

            version:
            serialset : sunitset;
                    { the system defined serial units }
            parallset : sunitset;
                    { the system defined parallel units }
            floppyset : unitset;
                    { those units that are on floppy drives }

            { this field is currently unused }
            floppydrive : packed array [Ø..7] of floppytype;
                    { type of floppy drive }

            reserved  : array [1..3Ø] of integer;

            {----- fields below this line are only on Winchesters -----}

            winchset  : unitset;
                    { those units that are on Winchester drives }

            { map of unitnumbers to drive and disk location & length }
            winchunit  : array [diskunits] of unitinfo;

            { the form the configuration record will take in memory }
            dynamic_config : configrec;

          end; {static_configrec}
```

```
{
                    System communication area (syscom)
                           and related types

          Needs definition of static_configrec, configrec, dirp.
           The needed definitions are in t_config, t_directry.

    (these types are used for pointers to these objects, so they could be
                       replaced with integers, i.e.
                          type dirp = integer; )
}


  {  declarations supporting idsearch / treesearch intrinsics --      }
  {  compiler using idsearch will have set up rw table with correct }
  {  len for rwinfo, and have set syscom^.rwtable to point to it.    }
    trsnodep = ^trsnode;              { symbol table node declaration }
    trsnode  = record                 { -- used by treesearch }
                 key   : alpha;
                 rlink : trsnodep;
                 llink : trsnodep;
               end;
    idsinfo  = record                 { idsearch returns results via this }
                 symcursor : 0..1023; { "pseudo record". compiler must    }
                 sy        : integer; { declare vars in this order and    }
                 op        : integer; { pass its symcursor to idsearch.   }
                 id        : alpha;
               end;
    rwtblrec = record
                 rwindex : array ['A'..'['] of integer;
                 rwinfo  : array [0..0] of
                             record
                               id : alpha;
                               sy : integer;
                               op : integer;
                             end;
               end  {rwtblrec};

                                      { SYSTEM COMMUNICATION AREA }
                                      { SEE INTERPRETERS...NOTE    }
                                      { THAT WE ASSUME BACKWARD    }
                                      { FIELD ALLOCATION IS DONE   }

    syscomrec = record case integer of
                  1 : ( boot_config : ^static_config
                                      { points to (temporary) location of
                                        the system configuration table at
                                        boot time, tables will be relocated
                                        later (in Initialize?) PLB });
```

```
  2 : ( config : ^configrec;
  unused : integer; { 1 spare word. }
  sysunit: integer;    { PHYSICAL UNIT OF BOOTLOAD }
  rwtable: ^rwtblrec; { reserved word table for treesearch }
  gdirp: dirp;         { GLOBAL DIR POINTER,SEE VOLSEARCH }
  diskinfo: packed record
              dseekrate: integer; {STEP RATE FOR DISK DRIVE}
              dreadrate: integer; {DISK READ COMMAND}
              dwriterate: integer;{DISK WRITE COMMAND}
            end;
  auxinfo:  packed record
              baudrates:  packed array [Ø..7] of Ø..15;
                          { 2 words, indices [Ø,4] not used }
              xonoff: packed array[Ø..7] of boolean;
              clockvalue: integer;  { tick clock rate }
              menudriven: boolean;  { using *system.menu }
              transparent: packed array[Ø..7] of boolean;
              { ignore special chars serial IO, no strip bit8}
            end;
  auxdata:  packed record
              spare7,spare6,spare5,spare4,
              spare3,spare2,spare1,spareØ: boolean;
            end;
  maxserports : Ø..7;
  expanstwo:  array [Ø..9] of integer;  {spares}
  auxcrtinfo: packed record
                verdlaychar: char;
                killqueue: char
              end;
  curtime : dp_integer; {hi,lo: integer}
  miscinfo: packed record
              nobreak,stupid,slowterm,
              hasxycrt,haslccrt,
              nointerrupts,hasclock: boolean;
              userkind:(normal, aquiz, booker, pquiz)
            end;
  crttype: integer;
  crtctrl: packed record
             rlf,ndfs,eraseeol,eraseeos,home,escape: char;
             backspace: char;
             fillcount: Ø..255;
             clearscreen, clearline: char;
             prefixed: packed array [Ø..8] of boolean
           end;
  crtinfo: packed record
             width,height: integer;
             right,left,down,up: char;
             badch,chardel,stop,break,flush,eof: char;
             altmode,linedel: char;
             backspace,etx,prefix: char;
             prefixed: packed array [Ø..13] of boolean
           end );
end { SYSCOM };
```

```
{
                    File Information Block
                       and related types

         Needs definition of vid, tid, direntry, fblksize.
           The needed definitions are all in t_directry.
}
                                        { FILE INFORMATION }

    closetype = (cnormal, clock, cpurge, ccrunch);
    fibp = ^fib;

    fib = record
            fwindow: windowp;     { USER WINDOW...F^, USED BY GET-PUT }
            feof,feoln: boolean;
            fstate: (fjandw,fneedchar,fgotchar);
            frecsize: integer;    { IN BYTES...Ø=>BLOCKFILE, 1=>CHARFILE }
            case fisopen: boolean of
              true: (fisblkd: boolean;   { FILE IS ON BLOCK DEVICE }
                     funit: integer;     { PHYSICAL UNIT # }
                     fvid: vid;          { VOLUME NAME }
                     freptcnt,           { # TIMES F^ VALID W/O GET }
                     fnxtblk,            { NEXT REL BLOCK TO IO }
                     fmaxblk: integer;   { MAX REL BLOCK ACCESSED }
                     fmodified:boolean;  { SET NEW DATE IN CLOSE }
                     fheader: direntry;  { COPY OF DISK DIR ENTRY }
                     flock : semaphore;  { File access lock. }
                     case fsoftbuf: boolean of { DISK GET-PUT STUFF }
                       true: (fnxtbyte,fmaxbyte: integer;
                              fbufchngd: boolean;
                              fbuffer: packed array [Ø..fblksize] of char))
          end { FIB } ;
```

```
{
                              User Work file stuff

                        Needs definition of fibp, vid, tid.
              The definitions can be found in t_directry, t_fileinfo.

}
     inforec = record
                 symfibp,codefibp: ^fib;         { WORKFILES FOR SCRATCH }
                 errsym,errblk,errnum: integer;  { ERROR STUFF IN EDIT }
                 slowterm,stupid: boolean;       { STUDENT PROGRAMMER ID!! }
                 altmode: char;                  { WASHOUT CHAR FOR COMPILER }
                 gotsym,gotcode: boolean;        { TITLES ARE MEANINGFUL }
                 workvid,symvid,codevid: vid;    { PERM&CUR WORKFILE VOLUMES }
                 worktid,symtid,codetid: tid;    { PERM&CUR WORKFILES TITLE }
               end { INFOREC } ;

{
                              System definitions

}
CONST
     osversion   = '[H3]'; {common os base}
     useCDint    = false; { whether to use carrier det interrupt for remote I/O}
     mmaxint     = 32767; { maximum integer value }

     has_timed_out = -1;
     not_in_time_q = -2;

     firstsysseg  = Ø;
     maxsysseg    = 127;
     firstuserseg = 128;
     maxuserseg   = 255;
     maxsubseg    = 15;

                    { THESE CONSTANTS USED BY I/O ROUTINES }

     cmaxport    =     1; {Ø..1}
     maxcard     =     1; {Ø..1}
     maxretry    =     4; { retry count for disk drivers }
     mievalue    =     1; { interrupt enable value }
     eol         =    13; { end-of-line ...ASCII cr }
     dle         =    16; { blank compression code }
     maxq        =    79; { type-ahead queue index limit }
     maxqpl      =    8Ø; { type-ahead queue length }
     xonqavail   =    6Ø; { number of characters available before xon sent }
     xoffqavail  =    2Ø; { number of characters available before xoff sent }
     xon         =    17; { control-Q transmitt on }
     xoff        =    19; { control-S transmitt off }
```

```
     hdiskaddr       = -912; { FC7Ø Winchester address }
     cond_hog        = false;

     hiiopriority  = 25Ø; { kbddriver (serial input) processes     }
     midiopriority = 245; { disk in/out, parallel out, serial out }
     lowiopriority = 24Ø; { lowest priority for system  processes }


TYPE

     byte        = Ø..255;

     iorsltwd = (inoerror,ibadblock,ibadunit,ibadmode,itimeout,
                 ilostunit,ilostfile,ibadtitle,inoroom,inounit,
                 inofile,idupfile,inotclosed,inotopen,ibadformat,
                 iwriteprot);

                                          { COMMAND STATES...SEE GETCMD }

     cmdstate = (haltinit,debugcall,
                 uprognou,uproguok,sysprog,
                 componly,compandgo,compdebug,
                 linkandgo,linkdebug);

                                          { CODE FILES USED IN GETCMD }

     sysfile = (adacomp,compiler,editor,filer,linker);

     integerp  = ^integer;
     bytearray = packed array [Ø..Ø] of byte;
     codeseg   = record case boolean of
                   true:  (int: packed array [Ø..Ø] of integer);
                   false: (byt: bytearray);
                 end;

     sibp = ^sib;
     sibvec = array [Ø..Ø] of sibp;
     sib = record { segment info block }
             segbase: ^codeseg;{ memory address of seg }
             segleng: integer; { # words in segment }
             segrefs: integer; { active calls - microcode maintained }
             segaddr: integer; { absolute disk address }
             segunit: integer; { physical disk unit }
             prevsp:  integerp;{ SP saved by getseg for relseg cut back }
           end { sib } ;
```

```
mscwp = ^mscw;
mscw = packed record  { mark stack control word }
         msstat: mscwp;   { lexical parent pointer }
         msdynl: mscwp;   { ptr to caller's mscw }
         msipc:  integer; { byte index in return code seg }
         msseg:  byte;    { seg # of caller code }
         msflag: byte
       end { mscw } ;

semp = ^semtrix;
tibp = ^tib;
tib = record { Task Information Block }
        regs: packed record
                waitq: tibp;          { QUEUE LINK FOR SEMAPHORES }
                prior: byte;          { TASK'S CPU PRIORITY }
                flags: byte;          { STATE FLAGS...NOT DEFINED YET }
                splow: integerp;      { LOWER STACK POINTER LIMIT }
                spupr: integerp;      { UPPER LIMIT ON STACK }
                sp: integerp;         { ACTUAL TOP-OF-STACK POINTER }
                mp: mscwp;            { ACTIVE PROCEDURE MSCW PTR }
                bp: mscwp;            { BASE ADDRESSING ENVIRONMENT PTR }
                ipc: integer;         { BYTE PTR IN CURRENT CODE SEG }
                segb: ^codeseg;       { PTR TO SEG CURRENTLY RUNNING }
                hangp: semp;          { WHICH TASK IS WAITING ON }
                iorslt : iorsltwd;    { Result of last I/O call. }
                sibs: ^sibvec         { ARRAY OF SIBS FOR 128..255 }
              end { REGS } ;
        maintask   : boolean;         { true if tib is root task (os tib) }
        startmscw  : mscwp;           { top mscw in task's stack          }
        nexttib    : tibp             { next pointer for list starting with}
      end { TIB } ;

decmax = integer[36];
longtrix = record case integer of
             Ø: (intar: array [Ø..Ø] of integer);
             1: (BCDar: packed array [Ø..Ø] of Ø..15)
           end {longtrix};

bytetrix = record case integer of
             1: (int   : integer);
             2: (byte  : packed array[Ø..1] of Ø..255)
           end;

memtrix = record case integer of
            1 : ( addr : integer );
            2 : ( loc  : integerp );
            3 : ( wp   : windowp );
            4 : ( int  : integer );
            5 : ( pack : packed array [Ø..1] of byte );
          end;
```

```
un_signed = record case integer of
              1 : ( i : integer );
              2 : ( p : integerp )
            end;

clocknode = record
              delay_sem  : semp;         { semaphore to signal to awaken }
              timed_out  : ^boolean;
              time_out   : dp_integer; { time to be awaken             }
              time_link  : ^clocknode; { points to next clocknode      }
            end;

segrange = firstsysseg..maxuserseg;
segsubrange = Ø..maxsubseg;
segpage = record
            diskinfo     : array [segsubrange] of
                             record
                               codeleng,
                               codeaddr : integer
                             end;
            segname      : array [segsubrange] of alpha;
            segkind      : array [segsubrange] of
                             (linked, hostseg, segproc, unitseg, seprtseg);
            textaddr     : array [segsubrange] of integer;
            seginfo      : array [segsubrange] of
                             packed record
                               segnum : segrange;
                               codeversion : Ø..255
                             end;
            notice       : string[79];
            codekind     : (static, vectored);
            lastseg      : integer;
            lastcodeblk : integer;
            filler       : packed array [1..56] of char
          end;

copierrec = record
              request_rendezvous : semaphore;
              end_rendezvous     : semaphore;
              copierbusy         : boolean;
              killcopy           : boolean;
              sunit              : integer;
              dunit              : integer
            end;
```

```
sys_control_word = packed record
                     filler    : Ø..255;          { bits 7..Ø   }
                     baud_rate : Ø..15;           { bits 11..8  }
                     drb       : Ø..15            { bits 15..12 }
                   end;


sys_stat_word =
    packed record
      case integer of
        1: (int : integer);
        2: (is_G_board,           { bit 15, G Board Identification               }
            bit14, bit13, bit12, bit11, bit1Ø,
                                  { bits 1Ø-14, ????                             }
            PERR,                 { bit 9, Parity Error                          }
            bit8,                 { bit 8                                        }
            MEMRTO,               { bit 7, Memory Reply Timeout                  }
            INTEN,                { bit 6, Interrupt Enable                      }
            EOB,                  { bit 5, EOB                                   }
            DINTR,                { bit 4, DINTR                                 }
            DDEN,                 { bit 3, Double Density Enable                 }
            BFFD,                 { bit 2, Boot From Floppy Disk                 }
            eight_inch,           { bit 1, indicates if 8" or 5.25" floppy       }
            IACKRTO               { bit Ø, Interrupt Acknowledge Reply Timeout}
                 : boolean)
      end;
```

```
{***************************************************************************

          THE REMAINING TYPE DECLARATIONS ARE FOR THE CONTROL OF
                      DISK, SERIAL, AND PARALLEL I/O

 ***************************************************************************}


    semtrix = record case integer of
                Ø: (sem: semaphore);
                1: (fakesem: record
                               count: integer; { outstanding signals }
                               waitq: tibp     { task queue }
                             end);
              end { sem } ;

    { for devices that use same reg for stat and cmd }
    statcmdrec = record case integer of
                   1 : ( bitØ : boolean );    { efficent way of testing bit Ø }
                   2 : ( command : integer );
                   3 : ( status : packed array[Ø..7] of boolean );
                 end;  { for devices that use same reg for stat and cmd }

    dstatrec = packed record case integer of {status / command reg}
                 1: (command  : integer);
                 2: (bow      : boolean;
                     dint     : boolean;
                     toi      : boolean;
                     tczi     : boolean;
                     icm      : boolean;
                     hbus     : boolean;
                     aece     : boolean;
                     busy     : boolean)
                 end;

    dmacntrbits = (RUN,DIE,TOIE,TCIE,ICM,HBUS,AECE,unused);
    dmarec = record   { DMA device register uses for both floppy and
                        winchester disk accesses }
               dcontrol : set of dmacntrbits;
               dstatus  : dstatrec;     {status / command reg}
               trcountl : integer;      {transfer count low order byte}
               trcounth : integer;      {transfer count high order byte}
               bufaddl  : integer;      {transfer buffer address low byte}
               bufaddh  : integer;      {transfer buffer address high byte}
               memex    : integer;
               intbase  : integer
             end;
```

```
fscomrec = packed record case integer of {status / command reg}
            { variants Ø..4 represent command bits }
            Ø: (allbitsint: integer);
            1: (command    : set of Ø..7);
            2: (commandint: Ø..255);
            3: (drivesel   : packed array[-8..7{Ø..3 are drives}] of boolean);
            4: (fillerØ    : boolean;
                filler1    : boolean;
                filler2    : boolean;
                filler3    : boolean;
                filler4    : boolean;
                filler5    : boolean;
                filler6    : boolean;
                filler7    : boolean;
                filler8    : boolean;
                filler9    : boolean;
                filler1Ø   : boolean;
                filler11   : boolean;
                unused1    : boolean;
                unused2    : boolean;
                densitysel: boolean;
                sidesel    : Ø..1);

            { the remaining variants represent status bits }
            5: (busy          : boolean;
                index         : boolean;
                trackØ        : boolean;
                bit3          : boolean;
                seekerror     : boolean;
                headloaded    : boolean;
                bit6          : boolean;
                notready      : boolean);
            6: (bitØ          : boolean;
                drq           : boolean;
                lostdata      : boolean;
                crcerror      : boolean;
                rnf           : boolean;
                writefault    : boolean;
                writeprotect : boolean;
                bit7          : boolean)
            end;
```

```
floppyrec = record {floppy device regiseter}
              fstatcom : fscomrec;{floppy status command register}
              track    : integer; {current track number. FDC updates this}
              sector   : integer; {sector to read or write}
              data     : integer; {for track to seek to. also data to read
                                   or write if no DMAC used}
              eobdintr : sys_stat_word; {has eob and dintr bits for modular}
              intprior : integer;
              flintbase: integer;
              filler   : integer;
              dma      : dmarec {floppy dma controller follows floppy regs}
            end;

taskfyle = packed record                          { for mass assignment }
             curblock     : integer;
             case integer of
               1 : ( curcylinder : integer );      { working copy here }
               2 : ( lo_cylinder : byte;
                     temp_hi_cyl : byte;
                     hi_cylinder : integer;
                     curhead     : Ø..7;
                     curdrive    : Ø..3;
                     secsize     : Ø..3;
                     zero_bits   : Ø..511 ); { these bits MUST be zeroed }
           end;

winchrec = record
             wdatareg   : integer;
             werrprecom : statcmdrec;
             wsectorcnt : integer;
             case integer of
               1 : ( taskfile : taskfyle );
               2 : ( wsectornum : integer;
                     wcyllow    : integer;
                     wcylhigh   : integer;
                     wsdh       : integer;
                     wstatcom   : packed record case integer of
                                    1: (command       : integer);
                                    2: (error         : boolean;
                                        unused1       : boolean;
                                        unused2       : boolean;
                                        datarequest   : boolean;
                                        seekcomplete  : boolean;
                                        writefault    : boolean;
                                        ready         : boolean;
                                        busy          : boolean)
                                  end;
                     dma        :  dmarec)
           end;
```

```
{  DISKBOARD AND DISKCONTROL ARE THE COMMUNICATION LINK BETWEEN DISKIO
   AND THE FLOPPY AND WINCHESTER DRIVERS }

diskboard  = record {disk control block for one floppy / winchester board}
               disksem     : semaphore; {attached to DMA/floppy interrupt}
               disktrix    : record case integer of {floppy/DMA device regs}
                               Ø : (address : integer);
                               1 : (floppy  : ^floppyrec);
                               2 : (hdisk   : ^winchrec)
                             end;
               dintreob    : ^sys_stat_word; { points to register on floppy
                                               board if modular otherwise
                                               it points to the system
                                               status word }
               {DENSITY reflects density last time drive was accessed}
               curdrive    : Ø..3;
               density     : packed array[Ø..3] of boolean;
             end;

modes = (readmode, writemode, clearmode);
diskcontrol = record {floppy control block for all floppy boards}
           disklock   : semtrix; {limits use of floppies to one at a time}
           unitselect: integer;   { reflects unit number }
           buffer     : windowp;   { transfer buffer pointer }
           boardnum   : Ø..maxcard;{ floppy board # for this access}
           board      : array[Ø..maxcard] of ^diskboard; {fields unique to
                                                    indiviual boards}
           mode       : modes;
           flags      : integer;   {2 = physical sector mode}
           trcount    : integer;   {bytes to transfer}
           inx        : integer;   {offset in fa^ to start transfer}
           startblock: integer;   {block to start transfer}
           ioerror    : iorsltwd;  {error result of tranfer}
           haswork    : semaphore; {signals driver to begin transfer}
           ready      : semaphore  {signaled when transfer complete}
        end;
```

```
{ COMMUNICATION LINK BETWEEN UNITREAD/UNITWRITE AND
  THE PARALLEL AND SERIAL DRIVERS }

iorequest = record {Comm link between I/O drivers and unitread/unitwrite}
              iohavework : semaphore; {tells driver to begin transfer}
              ioready    : semaphore; {locks port to one at a time use}
              iodone     : semaphore; {signalled when I/O complete}
              iounit     : integer;   {unit number for I/O}
              iowindowp  : windowp;   {points to buffer for transfer}
              ioflags    : integer;   {transfer mode bits}
              iobytes    : integer;   {number of bytes left to transfer}
              ioinx      : integer    {offset in iowindowp for tranfer}
            end;


          {**********      PARALLEL I/O TYPES      **********}

cards = Ø..maxcard;

paralrec = record {parallel port registers}
             porta    : statcmdrec;
             portb    : integer;
             portc    : statcmdrec;
             pcontrol : integer;
           end;

parcontrol = record {control block for parallel communication}
               paraltrix : record case integer of {parallel device registers}
                             1 : (pdevadd  : integer);
                             2 : (parallel : ^paralrec)
                           end;
               paronline : boolean; {true if unitclear found device online}
               parsem    : semaphore; {attached to parallel interrupt}
               request   : iorequest  {comm link to parallel output driver}
            end;
```

```
         {**********     SERIAL I/O TYPES      **********}

ports = Ø..cmaxport;

cntrl1bits = (DTR,RTS,RE,PE,ECHO,STOP1,BRK,NRML);
serialrec = record {serial port registers}
              data        : integer;     {data to be read or written}
              status      : packed record case integer of
                               1: (command      : integer);
                               2: (bitØ         : boolean);
                               3: (thre         : boolean;
                                   dr           : boolean;
                                   oe           : boolean;
                                   parityerr    : boolean;
                                   fe           : boolean;
                                   cd           : boolean;
                                   dsr          : boolean;
                                   dsc          : boolean)
                            end;
              control2    : integer;     {control register 2}
              control1    : set of cntrl1bits; {control register 2}
              filler      : integer;
              baudrate    : integer      {baudrate select on ME16ØØs}
            end;

auxsercntrl = record {control block for serial communication}
              qlock       : semaphore; {locks use of rear and front}
              havch       : semaphore; {kbddriver signals when it has char}
              writesem    : semaphore; {attached to serial output interrupt}
              writebell   : semaphore; {tells bellprocess to ring bell
                                        for input buffer overflow}
              readsem     : semaphore; {attached to serial input interrupt}
              rear        : integer;   {points to rear of input queue}
              front       : integer;   {points to front of input queue}
              chq         : packed array [Ø..maxq] of byte; {input queue}
              serialtrix  : record case integer of {serial device registers}
                              Ø: (sdevadd: integer);
                              1: (serial: ^serialrec)
                            end;
              stst        : semaphore; {stopped output wait on this}
              stwaitno    : integer;   {number of tasks w/ output stopped}
              fflag       : boolean;   { true means flush output }
              sflag       : boolean;   { true means freeze output }
              ioerror     : iorsltwd;  { kind of error during I/O }
              xofflag     : boolean    { true means xoff sent }
            end;
```

```
sercontrol = record {these Logically belong with auxsercntrl, but
                would upset sequence of U- global variables}
        0 avail         : integer;    { bytes available in input queue }
        1 statusq       : packed array [Ø..maxq] { true if error in    }
                              of boolean;         { read                }
        6 sendxoff      : semaphore; { tells writexoff to send an xoff
                                       char if xon/xoff enabled        }
        8 lport         : ports;     { port number for this record     }
        9 cardetsem     : semaphore; { signaled when carrier goes high}
       11 seronline     : boolean;   { port online when last checked   }
       12 request       : iorequest; { comm link between to driver     }
       23 auxport       : ^auxsercntrl {more fields like those in this
                                        record                         }
          end;

arraysercntrl = array[Ø..Ø] of sercontrol; {actual instance of this type
                                           may be less than cmaxport long}
```

```
VAR
   syscom    : ^syscomrec;          { MAGIC PARAM...SET UP IN BOOT }
   gfiles    : array [Ø..5] of fibp; { not used anywhere GLOBAL FILES, Ø=INPUT, 1=OUTPUT }
   userinfo  : inforec;             { WORK STUFF FOR COMPILER ETC }
   ostibp    : tibp;                { taskinfo block of op sys prog }
   emptyheap: ^integer;             { HEAP MARK FOR MEM MANAGING }
   inputfib,outputfib,              { CONSOLE FILES...GFILES ARE COPIES }
   systerm,swapfib: fibp;           { CONTROL AND SWAPSPACE FILES }
   syvid,dkvid: vid;                { SYSUNIT VOLID & DEFAULT VOLID }
   thedate   : daterec;             { TODAY...SET IF FILER OR SIGN ON }
   state     : cmdstate;            { FOR GETCOMMAND }
   heapinfo  : record               { heap management }
                 lock: semaphore;
                 topmark,
                 heaptop: integerp
               end { heapinfo } ;
   taskinfo  : record               { stuff for task management }
                 lock: semaphore;
             76  taskdone: semaphore; { signalled when task stops }
             77  ntasks: integer      { decremented when task stops }
               end { taskinfo } ;
   ipot      : array [Ø..4] of integer; { INTEGER POWERS OF TEN }
   filler    : string[41];          { NULLS FOR CARRIAGE DELAY }
   digits    : set of 'Ø'..'9';
   pl        : string;              { prompt line }
   chainname: string[23];           { chainer sets this - length > Ø means }
                                    { next getcmd executes chainname }
162 oldunitable : array [unitnum] of unitentry;  {27 unit descriptors to be
                                                   compatible with old programs.
                                                   Remaining unit descriptors
                                                   are in unitable which has
                                                   pointers to these entries as
                                                   well}
330 filename : array [sysfile] of string[23]; {'system.filer',etc.}
390 topofsibs: ^integer;
   safediskmode : boolean ;
392 oldport   : array [ports] of auxsercntrl; {two serial port control
                                               blocks to be compatible with
                                               old programs.  Remaining control
                                               blocks pointed to by serport
                                               which also points at these two
                                               blocks}
512 modular   : boolean;            {is currently an ME1600 computer}
513 maxunit   : integer;            {largest unit currently available}
514 unitable  : ^arrayunitable;     {name of vol, unittype, etc.}

     {.........Variable access by system U- programs ends here.........}
```

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

APPENDIX J: HARDWARE AND SOFTWARE CHANGES FOR III.Ø OPERATING SYSTEM VERSIONS GØ TO H3

Appendix J outlines the hardware and software changes that have occurred between version GØ and version H3 of the III.Ø Operating System. Section J-1 describes the changes from version GØ to HØ. Section J-2 discusses the changes from version HØ to H1. Section J-3 explains the improvements introduced in version H2, and Section J-4 outlines further improvements introduced in H3.

J-1. CHANGES FROM VERSION GØ TO HØ

Between versions GØ and HØ of the III.Ø Operating System, the hardware was upgraded to handle interrupts and Microms 14, 15, and 18 were installed.

The Operating System changed from noninterrupt to interrupt driven.

The HØ Compiler generated BNOT opcode instead of LNOT, which the GØ Compiler had generated. This change fixed problems such as ORD (NOT FALSE) returning a negative value.

J-2. CHANGES FROM VERSION HØ TO H1

For version H1, the code for the START process was enhanced to fix an inconsistency in process priority.

The number of user segments increased from seven to nine.

J-3. CHANGES FROM VERSION H1 TO H2

With version H2, the number of user segments increased from nine to 128.

The maximum code-segment size increased from 32K bytes to 64K bytes.

J-4. CHANGES FROM VERSION H2 TO H3

The H3 version supports a hardware upgrade for the ME1600 that allows software selectable floppy-disk density. This hardware upgrade is not required to run H3 software.

Software support that takes advantage of all the capabilities of the SB1600 is now available with the H3 release. Software support for the ME1600 with Winchester disk drives is also part of the H3 release; however, new PROMS that perform a "boot from Winchester disk" in the terminator card are required.

The H3 system provides a common bootable diskette for the ME1600, SB1600, and WD0900.

APPENDIX K. GLOSSARY

ARRAY

An ordered arrangement of characters, for example, a PACKED ARRAY OF CHAR.

BACKUP FILE

A copy of a file created for protection in case the primary file is destroyed unintentionally.

BAD BLOCK

A defective block on a storage medium, such as a disk, that produces a hardware error when attempting to read or write data in that block.

BASE SEGMENT

The portion of a segmented program that is always memory-resident.

BLOCK

A group of characters or bytes transmitted as a unit; one disk block of 512 bytes.

BOOLEAN VARIABLE

A variable which, when evaluated, produces either a true or false result.

BOOTSTRAP

A routine whose first instructions are sufficient to load the remainder of the routine and possible other routines into memory from an input device. Normally, it starts a complex system of programs.

BUFFER

A storage area used to hold information temporarily when it is being transferred between two devices or between a device and memory; often a specially designated area of memory.

CODE FILE

A file containing code to be executed; has the suffix of ".CODE".

COMMAND or COMMAND NAME

A word, mnemonic or character, by virtue of its syntax in a line of input, causes a predefined operation to be performed.

COMMAND STRING

A line of input that includes, generally, a command, one or more file specifications, and optional qualifiers.

COMPILE

The production of binary code (machine-readable) from symbolic instructions written in a high-level language.

COMPILER

Translates high-level language into machine code.

CONFIGURATION

A particular selection of hardware devices or software routines or programs that function together.

CONSOLE

The terminal that acts as the primary interface between the computer operator/user and the system; used to initiate and direct overall system operation.

CONSTANT

A value that remains the same throughout a distinct operation (as compared to a variable).

CONTROL CHARACTER

Controls an action rather that passing on data to a program.

CREATE

To open, write data to, and close a file for the first time.

DATA FILE

A file containing data to be manipulated by a program.

DEFAULT

The value of an argument, operand or field assumed by a program if a specific assignment is not specified by the user.

DEVICE

A hardware unit such as an I/O peripheral (disk, video terminal) - the physical unit as opposed to VOLUME, the logical unit.

DIRECTORY

A table that contains the names of, and pointers to, files on a mass-storage device.

DISASSEMBLER

A program that displays object code in human readable form.

EXPRESSION

A combination of commands and operands that can be evaluated to a distinct result.

FILE

A logical collection of data treated as a unit; may be work, code, text, foto or data file.

FILE SPECIFICATION

A name that identifies uniquely a file maintained in any system; must contain, at a minimum, the file name; may also contain the volume number and name.

FUNCTION

A routine that returns a value.

HEAP

An area of memory used for dynamic allocation. Pascal pointer variables are allocated from this area.

HEXADECIMAL

Whole numbers in positional notation using 16 as a base.

HIGH-LEVEL LANGUAGE

A problem-oriented language rather that a machine-oriented one.

INITIALIZE

Setting all hardware and software controls to starting values at the beginning of a new program.

INTERRUPT

The suspension of the normal programming routine to handle a sudden request for service. After completion of interrupt service, the program is resumed where it left off.

KEYBOARD ENTRY DEVICE

A device with a keybord (e.g., teletypewriter, video terminal) used by the system operator to control the system; CONSOLE.

LIBRARY

A collection of programs or subprograms contained as segments in a library file; normally contains frequently needed routines that may be accessed by other programs.

LISTING

A hard copy generated by a line printer.

LITERAL

The explicit representation of character strings or integers.

LOAD

To store a program or data in memory.

LOGICAL DEVICE NAME

An alphanumeric name assigned by the user to represent a physical device; used synonymously with the physical device name/number in the logical program.

MACHINE LANGUAGE

Instructions in binary code that can be operated on by the computer; as compared with high-level languages that can be read and understood by the user.

MAIN MEMORY

A set of storage locations connected directly to the processor.

NESTING

Routines enclosed within larger routines but not necessarily a part of the larger; a series of looping instructions may be nested.

OBJECT CODE

Relocatable machine-language code.

OBJECT PROGRAM

The source language program after it has been translated into machine language; output of the Compiler.

ON-LINE

Equipment and devices directly connected to, and controlled by, the central processing unit.

OVERLAY SEGMENT

A segment of code treated as a unit that can overlay code already in memory and be overlaid by other segments.

OVERLAY STRUCTURE

An overlay system consisting of a root segment and, optionally, one or more overlay segments.

PACK

To compress data in storage.

PROCEDURE
A routine that does not return a value.

QUALIFIER
A parameter specified in a command string that modifies some other parameter.

SOURCE LANGUAGE
A system of symbols and syntax easily understood by people that is used to describe a procedure that a computer can execute.

STACK
A block of successive memory locations accessible from one end on a LIFO basis (last-in-first-out).

SUBSCRIPT
A numerically valued expression or expression element that is appended to a variable name to uniquely identify elements of an array.

SWAPPING
Copying areas of memory to mass storage and back in order to use the memory for two or more purposes.

UTILITY
Any general-purpose program included in an operating system to perform common functions.

VARIABLE
The symbolic representation of a logical storage location that can contain a value that changes during a discrete processing operation; as compared to constant.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

PUBLICATION COMMENT FORM

We need your help!

This questionnaire is provided so that you, the user, can help us continually improve our customer support through accurate and easy-to-use documentation.

If you find any general or specific problems, please let us know. Even if you do not find problems, your evaluation of this document would be appreciated. Please send this questionnaire to:

THEMICROENGINE COMPANY
Subsidiary of Western Digital Corp.
2445 McCabe Way
Irvine, California 92714
ATTN: Product Documentation

Document Title: WESTERN DIGITAL UCSD PASCAL(TM) III.Ø
OPERATING SYSTEM REFERENCE MANUAL

Document No.: WD9893 Date: July 1982

YES NO Is this manual easy to read and understand?
----- -----

----- You (can, cannot) find things easily.
----- Language (is, is not) appropriate.
----- Organization (is, is not) logical
----- Technical terms (are, are not) defined.
----- Sentences and paragraphs (are, are not) coherent.
----- Other:

YES NO Are there enough examples?
----- -----

----- Examples (are, are not) (practical, workable, relevant.)
----- Examples (are, are not) adequately explained.
----- Other:

YES NO Do the charts, figures, and illustrations help you?
----- -----

----- Visuals (are, are not) (well designed, clear).
----- Other
----- Labels and captions (are, are not) clear.

YES ----- NO ----- Does the manual explain all you need to know?

What additional information do you need?

YES ----- NO ----- Is the information accurate?

----- (Does, does not) contradict your knowledge of the product. (List specific page and paragraph numbers below.)

----- (Does, does not) contain typographical errors. (List specific page and paragraph numbers below.)

In what ways do you use this document?

----- Learning to use the

----- Reference aid

----- Other:

----- Class instruction

----- Introduction to product

Any additional comments are appreciated. Thank you.

Name: ---------------------------- Title: ------------------------

Company: ------------------------- Division: --------------------

Address: -------------------------- City: ------------------------

State: ----- Zip: --------- Telephone: ------------ Date: ----------